Design, Implementation and Use of the MIRV Experimental Compiler for Computer Architecture Research

by

David Anthony Greene

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2003

Doctoral Committee:
Professor Trevor Mudge, Chair
Assistant Professor Todd Austin
Emeritus Professor Ronald Lomax
Assistant Professor Gary Tyson
Dr. Thomas Puzak, IBM Corporation

ABSTRACT

Design, Implementation and Use of the MIRV Experimental Compiler for Computer Architecture Research

by

David Anthony Greene

Chair: Trevor Mudge

This dissertation introduces MIRV, an experimental compiler developed for computer architecture research. We discuss the design and implementation of the compiler and use it to conduct studies of various techniques to tolerate memory latency. On the instruction side, a thorough examination of hardware and software prefetching techniques is performed to evaluate their utility on several modern computer designs. Various points of ambiguity in the literature are identified and the consequences of their specification are studied. A framework for describing software instruction prefetching algorithms is developed and extensions to current techniques are analyzed. Previous research has shown that larger data register sets than are currently available in modern microprocessors are desirable. Various extensions to this research are explored to further increase the utility of the register file.

To my family, with love and thanks.

ACKNOWLEDGEMENTS

Many people contributed to the success of this work and while I would like to acknowledge individually each by name, I would inevitably leave out deserving friends and relatives. Even the short list contained in these paragraphs is likely incomplete. I apologize in advance for such omissions and convey my deepened respect and admiration to all who contributed to the extraordinary experiences I have been fortunate to enjoy.

First and foremost I thank God – Father, Son and Spirit - for His guidance, strength and unconditional love. It is only with His aid that I have been able to complete this task.

My family has been a tremendous source of love, encouragement and inspiration. The support from my parents, Tom and Joleen Greene, and my sisters Amy Greene, Cathy Greene and Nancy Germanson kept me going not only through this specific task but through my entire life. I love all of you *so much*!

I also wish to thank Trevor Mudge, my advisor, and the entire dissertation committee for their insight and guidance. Trevor in particular had the grace to gently push this sometimes stubborn student forward without creating unnecessary tension and stress. His experience and wisdom helped me out of more than one jam.

The students at the University of Michigan provide tremendous support and stress relief. I worked for many years with Matt Postiff on the MIRV project and consider him to be my first sounding board and source of ideas. Likewise, Kris Flautner, Charles Lefurgy,

David Helder and Dave Oehmke contributed significant amounts of thought and effort to this project. Steve Raasch and I entered the program at the same time and have been great friends and collaborators since our first computer architecture course at UM. Troy Nolan also provided social outlets and humor at just the right times.

The spiritual support provided by St. Mary Student Parish in Ann Arbor, MI is invaluable. In particular I wish to thank Fr. Tom Firestone, Fr. Dennis Glasgow, S.J., Pat Waters, Anita Bohn, Gretchen Baumgardt, Anna Moreland and Mike Moreland for their individual attention to my spiritual direction and growth. I also wish to thank the many members of the St. Mary's Peace and Justice Commission and Rite of Christian Initiation of Adults (RCIA) program, in particular Derek Yip-Hoi, Arun D'Souza, Steve Coffman, Anita Chiappetta, O.P., Gene Poore, O.P. and Emily Malleis.

Most of my friends in Ann Arbor come from the St. Mary's Graduate Student/Young Professional Discussion Group. In addition to providing a wonderful intellectual and social environment, this group has taught me what it means to create friendships. Thank you so much to all of you wonderful people!

The REFRESH young adult diocesan retreat through the Catholic diocese of Lansing, MI filled a spiritual gap as I could never have imagined. I was fortunate enough to work with the planning team for three retreats. Thanks, guys, for all of your encouragement, patience and understanding.

Special thanks goes to Ken Laberteaux for graciously and judiciously providing guidance through his own experience of finishing his doctoral degree. Even more importantly, Ken has been a good friend.

My full-time job in Ann Arbor has been in the music scene (I was a graduate student in my spare time) and I owe much of that to that wonderfully wacky group known as The

Johnstown Cats swing band. The fabulous blend of Mike Sasena and trumpet, J. Dylan Clyne and Brody DeYoung on sax, Jeff Balcerski on trombone, Rob Felty on drums, Mark stock on guitar, Pete Klimecky on bass and the bang-up vox trio of Joe Mancuso, Chris Milas and Debbie Schooler provided just enough cover for this wayward tickler to keep the missed notes sufficiently hidden. Kudos also go out to former Cats Paul Forti, Hyatt Ford, Markus Nee, Scott Iekel-Johnson and Jeremy Welland. I have learned copious amounts of music theory through their pushing for more and better charts and I treasure every moment of it. Thanks guys for a wonderful, wonderful time!

Last but hardly least I must acknowledge my good and dear friends Mike and Jolene Sasena. Mike made the trek with me from the University of Notre Dame into enemy territory and our shared experiences in the ND marching band, The Johnstown Cats and at St. Mary's forged a bond like none other. Though I've only known Jolene a few years I feel like we are family. Both provided more support and encouragement than I could possibly catalog. You are very dear to me and I love you both!

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

LIST OF APPENDICES

CHAPTER 1

Introduction

1.1 The Thesis

The subject thesis of this dissertation is organized into three parts. The first concerns the MIRV compiler, a new research tool for computer architects and compiler developers. The second concerns the problem of instruction supply in modern computer systems and the third concerns the data supply problem in such machines.

1.1.1 MIRV: An Experimental C/C++ Compiler

Concerning the MIRV compiler:

- MIRV is an extensible and stable compilation tool-chain for computer architecture research.
- The design of MIRV leverages well-known software engineering practices and design patterns to present a modularized interface understandable by computer architects with rudimentary compiler background.

- MIRV provides an extensive set of regression testing and debugging tools that greatly eases the burden of compiler development for the computer architecture researcher.
- In terms of compiler correctness, MIRV outperforms several well-known research compilers currently available to computer architects in academia.

Research Contributions

We describe MIRV, an experimental C/C++ compiler for computer architecture research, in chapters 2, 3 and 4. MIRV has been designed to provide relatively easy accessibility to computer architects who may not be completely comfortable with compiler theory. It has been designed to provide a framework of pre-built components with which the researcher can construct new program analyses and transformations. The framework uses well-known software engineering design patters to provide a separation between program traversal, analysis and transformation.

The goal of the MIRV project has not initially focused on providing the best possible optimizing compiler around. Rather, the goal has been to provide a research platform at least as good as what is currently available. Chapter 5 presents some comparisons of MIRV to existing research compiler platforms.

Our work in compiler development has produced a set of research tools for computer architects. The MIRV compiler provides a framework for research into cooperative hardware/software design as illustrated in chapters 6, 7 and 8. Several novel features of the compiler tool-chain make it an attractive research platform.

The compiler operates in the traditional fashion, converting high-level source code into a high-level intermediate representation (IR). The high-level IR may be analyzed and transformed by a set of *filters*. A back-end phase converts the high-level IR into a lower-level

quad-based IR which may be manipulated by additional filters. A final linear pass converts the quads to machine assembly code.

While the overall operation of the compiler is traditional, several unique features enhance its capabilities in the research environment. The IR itself is a high-level prefix-form tree. The high-level form preserves most of the information available at the source code level, making certain analyses and transformations easier. For example, dominator and post-dominator computation is trivial given the structured tree form of the IR [1].

The prefix form of the tree allows the code generator to operate in a linear fashion as explained in section 2.6.1 of chapter 2. A simple LALR attribute grammar[1] is sufficient to generate code in a single pass without additional label patching passes.

The IR is also extensible. MIRV defines a *node attribute* interface for annotating information directly onto the IR tree. This is invaluable in the research environment because the compiler may mark points of interest in the program either for later phases of the compiler such as low-level code generation or it may embed information into the program binary which may be extracted by simulation software. This feature is used in the studies of chapters 7 and 8 to support speculative register allocation.

Chapter 3 presents a new software engineering design pattern called *Attribute Flow* for performing program dataflow analysis. This design pattern presents a hybrid of traditional iterative and more recent structural dataflow analysis algorithms. The high-level nature of the IR makes a structural dataflow engine desirable but deriving the dataflow equations for complex program structures can be challenging. Attribute Flow avoids this problem by iteratively applying structural dataflow equations at program control join points. The framework hides the details of the algorithm from the compiler developer, freeing him to

[1] for example, as created with a tool such as YACC[2]

concentrate of the representation of dataflow information and the actions necessary at the leaf nodes of the IR tree.

MIRV also includes a unified framework for intra- and inter-procedural program analysis and transformation. Operation in the inter-procedural domain uses the same intermediate form and analysis objects as in the intra-procedural domain. This is primarily provided through an intermediate-form linking phase in the compiler. This capability provides the researcher with a powerful tool for exploration of whole-program compilation without the need for complex link-time or post-link binary rewriting tools. The same familiar compilation model and environment is available in both the intra-procedural and inter-procedural modes of operation.

Finally, we have developed a suite of tools for automatic compiler bug characterization to ease the process of compiler debugging. These are described in chapter 5. The tools leverage the IR linking capabilities of the compiler to automate the bug localization and characterization process. The tools can capture the essence of both compile-time (compiler fault) and run-time (incorrect code generation) bugs. The compiler provides a rich set of command-line hooks to control the number, type and phase ordering of filters, a necessary feature for automated debugging.

In addition to bug characterization, MIRV provides over 700 regression tests and tools to incorporate larger source programs for stress testing. The regression framework has proven portable enough to validate the simulator software used in this work in addition to the compiler itself.

1.1.2 Instruction Prefetching

Concerning instruction supply:

- Previous work in instruction prefetching is underspecified and ambiguous.
- Aggressive sequential instruction prefetching outperforms the previous software instruction prefetching work studied in this dissertation.
- Some previous work on table-based instruction prefetching outperforms simplistic sequential prefetchers, but at a much greater hardware cost.
- New software instruction prefetching schemes proposed in this work slightly outperform sequential prefetching on the benchmarks studied.

Research Contributions

We study some existing instruction prefetching techniques and propose new techniques in chapter 6. During the course of this research we have discovered a number of deficiencies in the description of these techniques. Various points of ambiguity are identified and proposals made as to how to fill in the blanks. We have categorize these points of variance and explore the implied design space. Various combinations of these proposals are studied to determine the variants for which each technique performs best. We organize these points of variation into a *policy* framework that attempts to summarize how the instruction prefetchers studied operate. The studies of chapter 6 show that these points of variation can have a significant effect on the results observed.

Our examination of software instruction prefetching algorithms identifies several points of commonality. We generalize existing algorithms and develop a *filter* interface to specify how prefetch instructions are scheduled. This framework is formalized into a tuple-based specification that summarizes the operation of specific software instruction prefetching algorithms.

Two new software instruction prefetching techniques are proposed. *Compiler Hint Guided Prefetching* (CHGP) and *Cooperative Compiler Hint Guided Prefetching* (Cooperative CHGP) are shown to slightly outperform sequential prefetching on some architectures. The cooperative variant is found to be the better design.

1.1.3 Register Allocation

Concerning data supply:

- Previous work on hardware support for speculative register promotion is implementable on modern microprocessor systems.
- Such implementations do not degrade previously observed potential performance gains of speculative register promotion.
- The previous work can be extended in a straightforward manner to handle register allocation of potentially aliased data items.
- Register allocation of potentially aliased data items can dramatically improve program performance, though the impact is highly program-dependent.

Research Contributions

Chapters 7 and 8 present novel developments in register allocation. Chapter 7 explores previous work in speculative register promotion and verifies its implementability and utility on a modern microarchitecture. While the previous work mainly used an instruction counting argument to make its case, we verify the previously published results with cycle-accurate simulation. We discuss additional compiler considerations not covered by previous work, focusing on impacts on the analysis and transformation phases. We develop an Application

Binary Interface (ABI) for the Store-Load Address Table (SLAT), the hardware mechanism used to support speculative register promotion.

In chapter 8 we develop extensions to the existing register renaming hardware to support speculative register promotion of potentially aliased data. Our extensions modify the processor register renaming hardware to allow register allocation of potentially aliased data. While only one benchmark benefits from this technique, the improvement is quite dramatic, leading us to conclude that the extensions may be applicable to other programs as well.

1.2 Benchmarks

In this section we state our assumptions about the benchmarks used throughout this dissertation. It is practically impossible to fully state all of the assumptions made. In our case we have attempted to state as fully as possible the design parameters used. Because stating the full set is difficult, our goal is to release all of the code used in these studies so that other researchers may benefit from it. This includes the MIRV compiler source, M5 simulator source and benchmark datasets if possible[2].

Throughout this dissertation we use the SPEC 95 and 2000 benchmark suites in our studies. Most of our studies used the reduced data sets listed in table 1.1. The table includes information about the number of dynamic instructions executed and the instruction memory footprint for benchmarks used in the prefetching studies of chapter 6. We verified some of the data in chapter 6 against the larger SPEC test inputs to confirm that the smaller data sets did not impact the prefetching results observed. We note that some of the results (particularly those in chapters 7 and 8 are missing data for some of the benchmarks. These experiments did not complete in a timely fashion and thus were not available for inclusion.

[2]Licensing restrictions prevent us releasing the benchmarks themselves and some of the data set files.

We include the partial results to maintain consistency among the benchmark sets presented and to provide as much information as possible.

1.3 Simulation Environment

Throughout this dissertation we use the M5 simulation environment [3]. M5 is a cycle-accurate, event-driven simulator developed at the University of Michigan. In provides a detailed pipelined, out-of-order superscalar processor model that includes simulation of memory hierarchies with bus contention. M5 also includes device drivers to allow full system execution with operating system effects though our studies did not include this modeling due to the lack of an operating system for our experimental instruction set architecture. Simulation model parameters are included in the chapters relevant to each study performed.

Benchmark	Suite	Arguments	Dataset	Insn.	Size
compress	SPEC 95				
gcc	SPEC 95	`-quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O`	`regclass.i` from test	141 M	654 KB
go	SPEC 95	`9 9 null.in`	ref input		
ijpeg	SPEC 95	`-image_file specmun.ppm -compression.quality 25 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp`	test input		
li	SPEC 95	`boyerExit.lsp`			
m88ksim	SPEC 95	`-c < ctl.lit`	train input		
perl	SPEC 95	`< jumble.in`	jumble.pl	115 M	107 KB
vortex	SPEC 95	`vortex.lit`			
ammp	SPEC 2000	`< bughunt.in`			
art	SPEC 2000	`-scanfile c756hel.in -trainfile1 a10.img -stride 2 -startx 134 -starty 220 -endx 139 -endy 225 -objects 1`	test input		
bzip2	SPEC 2000	`input.random 1`	test input		
gcc	SPEC 2000	`bughunt.i -o gcc00.s.log`			
gzip	SPEC 2000	`input.compressed 1`	test input		
mcf	SPEC 2000	`inp.in`	test input		
mesa	SPEC 2000	`-frames 1 -meshfile mesa.in -ppmfile mesa.ppm`	test input		
parser	SPEC 2000	`2.1.dict -batch`	test input		
equake	SPEC 2000	`< inp.in`	test input		
vortex	SPEC 2000	`lendian.raw`	reduced	169 M	303 KB

Table 1.1: SPEC Reduced Datasets

CHAPTER 2

The MIRV C/C++ Experimental Research Compiler

2.1 Introduction

This chapter describes the MIRV C/C++ research compiler. Postiff's dissertation describes the compiler back-end in detail [4]. Therefore, we concentrate on the front-end design. We begin with an overview of the MIRV compilation model in section 2.2. Section 2.3 presents the MIRV language and intermediate format used by the front-end analysis and transformation passes. We describe not only the syntax and semantics but also the internal representation as seen by the programmer in section 2.4. Section 2.5 provides a brief overview of the back-end and low-level code generation process. Special language support required for C and C++ is surveyed in section 2.6 while section 2.7 mentions some other compiler frameworks described in the literature.

Where appropriate, each section presents not only the MIRV design as it currently exists, but also suggestions for future improvements. As with any large software projects, lessons are learned along the way. MIRV is no exception and whether due to programming environment or design limitations, the compiler framework can certainly be improved in several areas.

MIRV is an experimental research compiler that we have developed over several years. It compiles programs written in both C and C++ and provides full automatic template instantiation and integration with popular simulation environments such as SimpleScalar [5] and M5 [3]. Back-end targets exist for SimpleScalar/PISA (a MIPS derivative ISA), Intel IA32, ARM and some experimental research instruction set architectures.

2.2 Compilation Model

In this section we present the compilation model used in MIRV. The compilation model describes the internal representation of source programs in the compiler and the sequence of actions needed to transform the source into another language (in this case, MIRV and low-level assembly code). The compilation model also presents the Application Programmer Interface (API) available to the compiler designer. The API specifies how one might go about integrating new source language front-ends, analyses and transformations into the compiler. This section concentrates on a broad overview of the "MIRV process." Later sections present in-depth descriptions and API specifications for programming the compiler front-end.

Figure 2.1 presents an overview of the compilation process. The left side presents the traditional mechanism as implemented in MIRV. Multiple source files are compiled individually by the front-end. Each file is translated to a high-level *intermediate representation* (IR) and processed by *filters* in turn. After the front-end processes each file, the back-end is invoked to perform low-level transformations and generate assembly code. The assembler is invoked to create an object module. All object modules are then linked together to produce an executable.

The right side of the figure shows some non-traditional mechanisms available in MIRV.

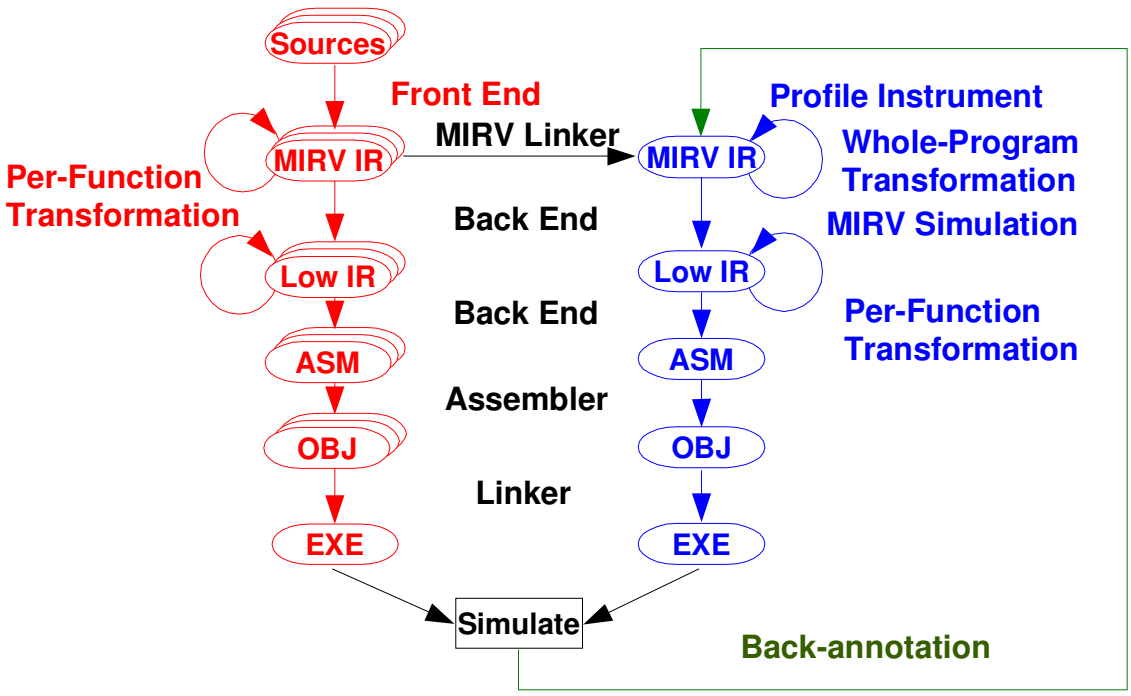


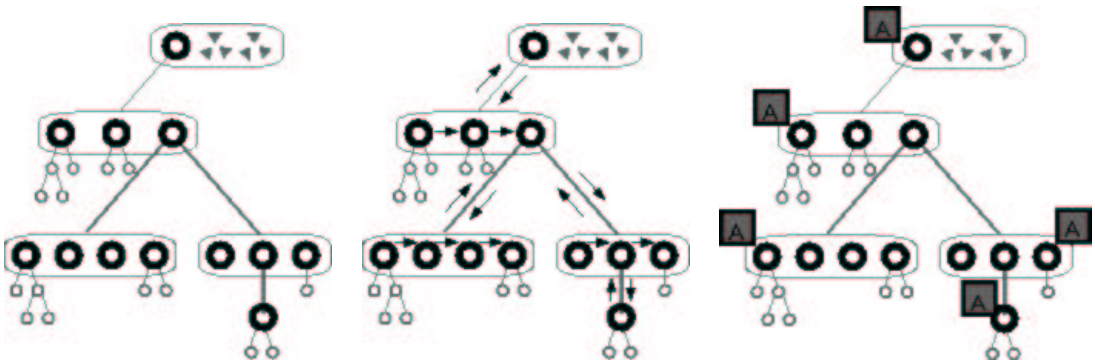
Figure 2.1: MIRV Compilation Flow

The high-level intermediate representation can be linked together to create one high-level representation of the entire source program. This allows whole-program analysis and transformation. This representation facilitates inter-procedural analysis and optimizations such as call graph construction and function in-lining. In addition, profile information from a previous run of the program can be automatically back-annotated into the representation[1].

Whole-program analysis has been an active area of research for some time. Many of these studies were performed on object code after program linking [6]. By allowing program linking using the high-level intermediate representation, MIRV provides a consistent environment for performing both intra- and inter-procedural analysis and optimization. We refer to this process as *source-level linking* even though technically the linking occurs within the intermediate representation[2]. This linking process also provides several advantages dur-

[1] Instrumentation of the program need not require a MIRV-linked program. Such a representation is not needed for back-annotation either, though that is the procedure used in our current profiling filters.

[2] *Linking* in this case refers to the process of resolving external symbol references to the symbol objects



(a) Tree Structure (b) Attribute Propagation (c) Attribute Annotation

Figure 2.2: MIRV Compilation Model

ing the debug process due to the consistent interface (the MIRV IR file) available to the debugging tools.

Operation Within a Translation Unit

A *translation unit* within the context of MIRV is defined as a single file to which translation and transformations are applied. This can be a single high-level C or C++ source file, a pre-translated MIRV IR file or a whole program if source-level linking has been applied.

At the translation unit level, the compiler consists of a set of filters that operate on the intermediate representation of the program. The intermediate representation is an operator tree representing the MIRV language where every node may contain a set of user- and compiler-defined attributes. The attributes are usually computed and used by the filters that are invoked on the tree. Successive passes of filters communicate using these node attributes. Figure 2.2 illustrates the pieces in the compilation process. The generic tree structure of MIRV is shown in figure 2.2(a). There are two main groups of attributes: *parse*

stored within the MIRV compiler. As the high-level form has no concept of a branch *per se*, label patching need not be performed.

attributes and *node attributes*. Parse attributes are the *synthesized attributes* and *inherited attributes* that are passed up and down a parse tree during traversal as shown in figure 2.2(b) [7]. *Node attributes* are pieces of information that are associated with nodes in the operator tree. Figure 2.2(c) shows an annotated MIRV tree. Note that annotations can appear anywhere on the tree, not just at the leaves.

2.2.1 Filters

Filters are categorized according to their purpose into three groups. *Analysis filters* traverse the tree in some order and perform computation using parse attributes that are propagated during tree traversal. The results of this computation are represented as node attributes on the operator tree. *Transformation filters* change the structure of the tree by adding and removing nodes. How a transformation filter changes the tree structure is usually determined by the node attributes computed by analysis filters. *Snapshot filters* neither set node attributes, nor do they alter the structure of the existing tree; they simply traverse the tree and invoke methods on external objects based on the operators and node attributes in the tree. Examples of snapshot filters include the linearizing or "pickling" filter used to print the intermediate form to a file and the high-level MIRV IR simulator [8]. By coordinating the order in which these three types of filters are run, a program can be optimized and translated to a target machine. The level of optimization can be varied by adding and removing filters and changing the order in which filters are run. Filters declare dependencies on various analyses through the attribute system. Therefore, the user need not be concerned about such dependencies when reordering analysis and transformation filters. All filters can be invoked directly from the compiler command-line, another key feature that improves the utility of the tools we discuss below.

2.2.2 Attributes and the Compilation Process

Dependencies between filters exist as the dependence of a filter on a set of node attributes. In other words, a filter does not explicitly specify what other filters must precede its execution but rather states the names of the node attributes it is dependent upon. One use of such dependencies in the compiler is the caching of analysis attributes. Because filters declare which attributes are modified by a program analysis or transformation and which attributes are needed as input to a filter, the compiler can know when to re-invoke a full analysis pass for a transformation filter. Currently, any transformation which alters the program structures is assumed to invalidate all dataflow attributes, though special interfaces exist to allow the filter designer to bypass some of these assumptions for trivial transformations (expression reassociation, for example) [3]

Attributes are internally represented in one of five forms. The simplest is the boolean attribute, represented with the C++ `bool` type. The `int` type is used to represent integer attributes. Similarly, double attributes are represented by the `double` type. A C++ `std::string` represents string attributes. Finally, a `mirvNodeAttribute` abstract base class is provided to allow filter designers to create their own types of attributes. Only the first four attributes are directly supported by the MIRV language as representing attributes derived from `mirvNodeAttribute` would require an extensible parser to read in a MIRV IR file.

[3] Currently such dependencies are expressed with an explicit query into the attribute database to see which attributes have been invalidated. A more automatic approach to this problem is an area of future work.

2.3 Intermediate Representation Language

In any compiler, the program representation format has a great impact on the analysis and transformation approaches used. For example, iterative dataflow analysis usually implies a low-level basic-block pseudoinstruction representation, while structural dataflow analysis requires high-level information about the control structure of the program [9, 10, 11]. In addition, there is a tension between the desire to know as much information about the program and the desire to expose as much of the computation abstracted by the high-level language as possible to allow effective optimization.

The MIRV compiler front-end uses a tree representation of the MIRV language. The MIRV language is intended to be a generic high-level intermediate language that can be targeted by many different high-level source languages, in the spirit of compilers that use a common analysis format for several high-level languages [12]. The tree is in prefix form, which aids the syntax-directed translation scheme used in the back-end low-level intermediate representation generator by providing proper context to the parser about what is to be expected next in the input stream.

The language itself is quite similar to a sanitized version of the C language, other than its prefix form. Operators exist not only for expression trees, but also loops, switch statements, if-else constructs and so forth. The key difference between this representation and the representation used by most compilers is that the high-level control structure of the program is preserved. While most compilers must build a control graph from a basic block representation to perform some types of analysis, the control structure is implied by the structure of the MIRV language. This structure is available right up to low-level intermediate code generation time, allowing transformations such as loop unrolling and strength reduction with test replacement without the need to re-synthesize important information.

Some transformations need to see a large context which is lost during low-level instruction quad generation. Expression reassociation, for example, loses much of its power if the expressions are broken into the two-source, one-destination format typical of low-level quad representations. Other transformations like loop-invariant code motion can make larger motions in a single pass if such context is preserved.

Expressions

In MIRV, an *expression* is a computation that does not modify program state. It simply uses available data to perform some arithmetic or reference other data. In particular, unlike in the C language, function calls and assignments are not expressions because they (potentially) modify program state through side-effects.

Type of Operation	**MIRV Structure**	**Description**
Direct Reference	*op data*	*op* := `vref`, `cref` or `fref`
	`aref` *data index-list*	Array reference
	`vfref` *aggregate field*	Field reference
Indirect Reference	`airef` *expr index-list*	Indirect array reference
	`vfiref` *expr field*	Indirect field reference
Address	`addrOf` *expr*	Take the address
Size	`sizeOf` *type*	The size in bytes of *type*
Arithmetic	*op expr expr*	*op* := `add`, `sub`, `mul`, `div`, `mod`, `pow` or `sqrt`
	`neg` *expr*	Negation
Bitwise	*op expr expr*	*op* := `and`, `or`, `xor`, `shl`, `shr`, `rol`, `ror` or `xor`
	`cpl` *expr*	Complement
Boolean	*op expr expr*	*op* := `cand`, `cor`, `lt`, `le`, `eq`, `ne`, `ge` or `gt`
	`not` *expr*	
Casting	`cast` *type expr*	Cast *expr* to *type*
Literal	`lit` *text* `ulit` *text expr* `blit` *text expr expr*	*text* := `printf`-style format string

Table 2.1: MIRV Expressions

Table 2.1 presents the expression operators in the MIRV language. The arithmetic operators are fairly standard. The `pow` and `sqrt` operators were included to support ma-

chines with instructions to perform these operations. This was somewhat arbitrary, as one could make a case for including operators such as `sin` and `cos`. However, because the C language encapsulates `pow`, `sqrt`, `sin`, `cos` and many other complex operations in math library functions, these operators are not used in practice[4].

The logical `cand` and `cor` operators are analogous to the C `&&` and `||` operators with one significant difference: they do not short-circuit for the purposes of dataflow analysis. This property allows the compiler to separate program control flow from logical evaluation, simplifying the dataflow model. To preserve the C semantics, the compiler front-end translates the short-circuiting operations to equivalent control flow constructs. In the case where it can prove that side-effects to not exist, the cand and cor operators are used directly, avoiding unnecessary branches. The current back-end does preserve the C semantics in these cases even though correctness does not require it. This provides a significant savings in dynamic instruction count.

The casting operation is used only to maintain type consistency throughout the program. Where C semantics require an implicit cast, the MIRV front-end inserts an explicit cast. This simplifies the code generator by removing the burden of type comparison and cast insertion.

There are six operators to reference various types of data. A `vref` includes an operand specifying a specific variable in the MIRV program. Variable names are globally unique. An `aref` works exactly like the C subscript operator. Arguments are an array identifier and index expressions. An `airef` is an array index off a pointer variable (i.e. pointer arithmetic). A `vfref` references a field from an aggregate type while a `vfiref` references a field from a pointer to an aggregate (like C's `>` operator).

[4]The back-end is able to transform some of these function calls into machine instruction sequences.

Pointer arithmetic is allowed, but there is no implicit scaling as in C. To describe the scaling in a machine-independent manner, the `sizeOf` operator is provided. Given a type identifier, `sizeOf` returns the size in bytes of an object of that type. Note that the size is not necessarily known until the machine-specific code generator examines the type due to padding requirements of a particular Application Binary Interface.

MIRV provides a high-level interface to low-level machine-dependent code through the literal expression operators. The `lit` operator simply passes a raw string to the back-end. The string may contain `print`-style format specifiers. If the literal is an immediate child of an assignment statement the assignment destination can fill in the format placeholder. Back-end support must exist to fill in the proper value for the placeholder. The current PISA back-end understands how to fill in constant values and machine register names. After processing the string should be in a form the assembler can understand. A common use is to embed bytes directly into the produced assembly file using ASCII hexadecimal notation. The GNU assembler will directly translate these to raw bytes in the object file. The `ulit` and `blit` operators extend the power of literal expressions by providing an interface to attach subexpressions and write their results into a format string containing more place-holders, one per child expression and an optional placeholder for an assignment right-hand-side.

Literal expressions can participate in dataflow analysis in a limited fashion. The analysis routines will not understand the semantics of the operation but by judiciously using `ulit` and `blit` operators and positioning them as children of an assignment operator that specifies any store semantics of the operation the programmer can often incorporate the literal operations into the dataflow engine without trouble.

MIRV Statements	**Description**
`assign` *expr expr*	Assignment
`call` *function arg-list*	Function call
`ficall` *expr expr-list*	Indirect function call
`if` *cond then*	If-then
`ifElse` *cond then else*	If-then-else
`switch` *cond case-list* `case` *constant body*	Multiway branch Multiway branch target
`while` *init cond body incr*	Like C for/while-loop
`doWhile` *init cond body incr*	Execute *body* and *incr* at least once
return	Function return
`destBefore` *label*	Setup context for gotoDest branch before block
`destAfter` *label*	Setup context for gotoDest branch after block
gotoDest *label*	Branch to destBefore or destAfter
`goto` *label*	Branch to an arbitrary target
label`:`	`goto` target

Table 2.2: MIRV Statements

Statements

Table 2.2 lists the statement structures in the MIRV language. In MIRV, a *statement* is any piece of code that can potentially modify program state. Thus function calls and assignments are grouped under the statement heading. State-modifying structures cannot be nested arbitrarily within expressions as they are in C. This greatly simplifies program analysis by cleanly separating code that modifies state and code that uses it. This separation allows more efficient dataflow computation in some circumstances as described in section 3.2. The one exception to the separation rule is assignment of function call return values because there must be some method of communicating the call result to the target variable. Thus a function call can only appear as its own statement or as the immediate right-hand-side of an assignment statement.

The ubiquitous `assign` statement needs little explanation. We group it under the statement category because it modifies program state. Disallowing nested assignments within expressions greatly simplifies the design of dataflow passes by clearly sequencing the order of operations. With nested assignments as in C, the order of evaluation is often

implementation-defined. The MIRV front-end defines the order for the filter designer. This ordering does not inhibit program transformation because dataflow analysis can determine when assignments are independent.

The `fcall` and `ficall` operations invoke a function by name or through a pointer, respectively. Functions may only appear as single statements, or as a single expression in an assignment statement. This simplifies the design of dataflow analyzers because function calls often complicate the analysis by potentially changing program state. Calls are guaranteed only to appear in certain situations, which eases the burden on the designer. This rule also implies that a function call cannot be nested as an argument to another function call. This greatly simplifies generation of the call-stack manipulation code in the back-end.

For the most part, MIRV strives to represent the input program control flow in a well-structured form. A control-flow graph $G = \langle N, E \rangle$ is *well-structured* if it can be represented as a set of *forward* edges N_f and *back* edges E_b such that $\langle N, E_f \rangle$ forms a directed acyclic graph in which every node is reachable from the entry node and all the edges in E_b are *back edges*—edges whose heads dominate their tails[5]. In simpler terms, all loops in the graph must be natural loops characterized by their back edges—each loop body may only be entered from its header node. For example, an `if-then-else` construct has this property as do C `while` loops if there are no `goto` target labels in their bodies. The high-level MIRV representation simplifies program analysis by preserving high-level control-flow information such as loops. Structural dataflow algorithms can be used on such programs to more quickly compute problem solutions [11, 10] and allow simpler incremental updating of dataflow information when program structure changes [6].

The decision constructs `if`, `ifElse` and `switch`/`case` have the standard C functionality.

[5] A control-graph node d *dominates* node i if every execution path from the entry node to i includes d.

[6] Although a prototype structural dataflow engine was developed for MIRV, it is not currently in use because the iterative algorithm is simpler and is sufficiently fast for our purposes.

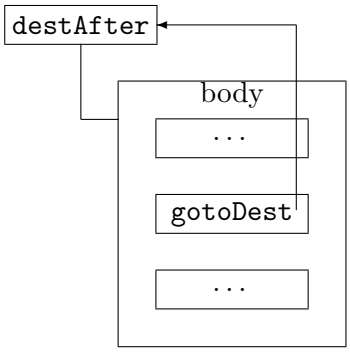
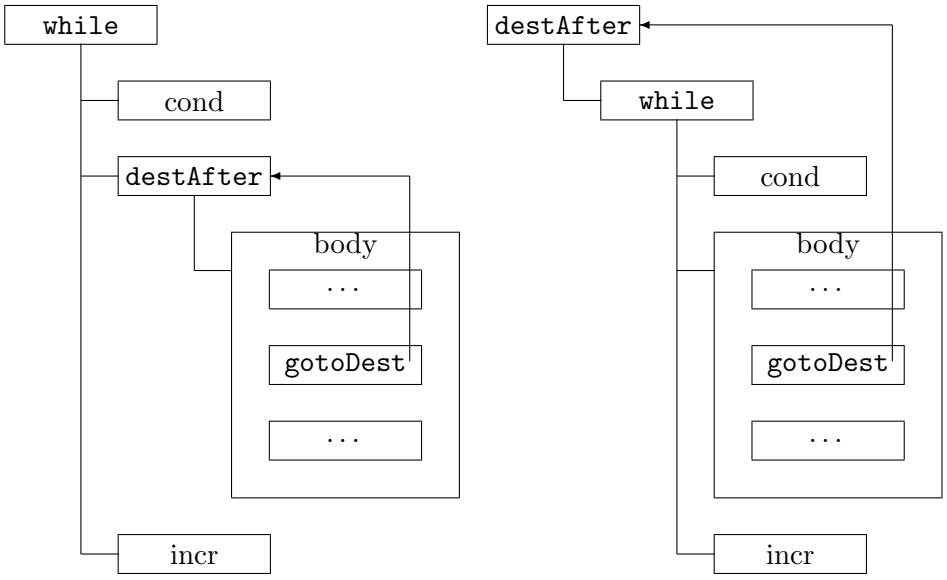


Figure 2.3: gotoDest Structure

The two looping constructs operate as C `while` and `do` loops.

While C's `goto` statement can easily render a function unstructured, some forms of goto are particularly useful. A goto out of a deeply nested loop, a break out of a switch construct, or a C `next` or `continue` operation are all structured goto forms. These are supported through use of the `destBefore`/`destAfter`/`gotoDest` structure. A `destBefore` or `destAfter` describes a label before or after a block of code, respectively. A `gotoDest` inside the enclosed block implies an unconditional transfer of control to the label. The gotoDest includes a label argument detailing which label to jump to. The `gotoDest` target must enclose the block containing the `gotoDest`, as shown in Figure 2.3. This maintains structured control. Figure 2.4 shows how the `destBefore`/`destAfter`/`gotoDest` structure can be used to implement the C control structures described above.

MIRV supports non-structured control flow using a `goto` construct. If an input program makes use of the C `goto` statement, MIRV attempts to transform it into an equivalent structured form using `destBefore` and `destAfter` statements [13]. This greatly simplifies program analysis. Our experience is that most programs are written in a structured form, because it allows easier program maintenance. Those few programs utilizing constructs such



(a) `continue` (b) `break`

Figure 2.4: C Control with `gotoDest`

as arbitrary gotos can usually be re-written easily.

Node Attributes

As mentioned in section 2.2.2, node attributes are used to convey information from one filter to another during the compilation process. By extending the concept of "filter" we can treat the back-end code generator as another filter pass through the MIRV intermediate representation. To support this idea, the MIRV language includes facilities for writing out some kinds of node attributes to the linearized on-disk representation. We call these *linearizable attributes*. Specifically, boolean, integer, double and string attributes are directly supported by the MIRV language definition. Attributes derived from `mirvNodeAttribute` must be converted to one of the first four forms to be dumped to the MIRV IR file. The most likely candidate is a string attribute. Any system reading in the MIRV IR file would need

```
attribute {
  name          "target_flags"
  register      true
  used          true
  addrof        false
  temp          false
}
vdecl export unid target_flags unid sint32 {    # sint32
}
```

Figure 2.5: Attribute Representation in the MIRV IR

a parser to convert the string attribute back to an appropriate in-memory data structure.

In the linearized form, attributes can appear almost anywhere. The general form is a brace-delimited list of ⟨ attribute-name, attribute-value ⟩ pairs, as shown in figure 2.5. In this example, a variable declaration is preceded by a block specifying the attributes of the variable being declared. This particular variable has a name[7] and several boolean values describing whether the variable can be put into a register, whether it is actually used in the program, whether its address was taken and whether it is a compiler-generated temporary for holding expression computation results. Note that the attribute values completely specify the type of attribute being described so that separate keywords for specifying attribute types are not required.

2.3.1 Future Improvements

Attributes may appear before any declaration (type, variable, etc.), any statement and certain kinds of expression (data reference expressions and loop condition expressions). We have not yet found any practical use for annotating arbitrary expressions with attributes though there is conceptually nothing that disallows it. Our current implementation does not provide support in the parser for recognizing such annotations, though such support

[7]Name attributes are deprecated in the current version of MIRV since the unid string completely specifies the necessary information.

can easily be added.

Future Directions

The multitude of data referencing operators can complicate the program analysis code. Transformation filters tend to get cluttered with actions to handle all of the referencing operations. Often these actions could be folded into a generic algorithm. For example, rather than using a multi-level `deref` tree, a single data reference with an "indirection level" could specify the same operation more compactly. Such a scheme is used in the back-end `DataDescriptor` (c.f. section 2.5.2) with great success, though it may complicate the alias analysis somewhat by hiding nested dereference operations.

In addition, `afref`s, `airef`s, `vfref`s and `vfiref`s hide addressing computations from the optimizer. There is an inherent tension between preserving the structure and array access information and exposing the calculations using pointer arithmetic. With the array reference format, there is more information about which specific element is being accessed. Alias analysis, however, can recover some of this information. For the time being, we provide a transformation that converts array operations to equivalent pointer arithmetic. An `offsetOf` operator may be provided in the future to allow exposure of complex aggregate field reference arithmetic. This will become more useful in the context of C++ multiple and virtual inheritance as `this` pointer adjustments are required to address sub-objects within the aggregate.

The looping constructs used to include blocks to initialize and increment the loop control variable, *a la* C's `for` loop. We had hoped that these statements in the looping constructs might provide more high-level information about the loop, but this proved not to be the case for our current analyses. Analyzing the program to place the correct iteration construct in

this statement is equivalent to performing the analysis necessary for some transformation (strength reduction and test replacement, for example) that use the iteration information. Thus, these clauses were removed. In addition, due to the restrictions on placement of state-modifying code, complex loop condition code (function calls, for example) must be duplicated before the loop and within the loop body. To address this problem, pre- and post-condition blocks (executed before and after each evaluation of the loop condition, respectively) will appear in a future version of MIRV. These blocks will allow conditions containing function calls (or other complex code) to be constructed without static code duplication.

At this point in MIRV's evolution, the language is capable of expressing native C constructs while preserving most of the high-level control information. C++ adds new information, much of which cannot be expressed in a C-like language without losing significant information. Inheritance hierarchies, for example, are useful for performing type analysis and converting virtual function calls into direct method calls [14]. It may be possible to express such hierarchies with attributes, similarly to the way the static call graph is presented to the back-end. Exploring this and similar aspects of the MIRV language design is a large area of future work.

Conclusion

The MIRV language has been designed to be simple—anyone with a background in C or a similar language can easily grasp its meaning–and to simplify program analysis and code generation. The structured form of MIRV means transformations such as loop unrolling and strength reduction do not need to rediscover relevant program control structures. When program control flow cannot be represented structurally, a `goto` construct is used. The prefix

nature of MIRV makes a syntax-directed translation of MIRV into a lower-level intermediate form almost trivial.

2.4 Intermediate Representation API

Once armed with the language definition as presented in section 2.3, the compiler designer must know how the language is represented in-memory and the tools available to manipulate it. This section presents the programmer view of the MIRV language. We present the MIRV class hierarchy and important methods for manipulating MIRV trees.

Class Hierarchy

Because the MIRV grammar models a prefix-form linearized tree, it is natural for the compiler to construct an in-memory tree representation of a source program. Because MIRV is relatively high-level in nature, the data structures are almost a one-to-one mapping to the MIRV grammar and even to a grammar for a language like C.

Just as the MIRV grammar partitions constructs into statements and expressions, so to does the MIRV data structure API. This partitioning simplifies a number of filters as most analyses and transformations are only concerned with a subset of MIRV tree constructs. In fact, the MIRV class hierarchy not only achieves the classical object-oriented programming goal of sharing interfaces and implementations but also provides a classification framework for various code structures in the same way grammar non-terminals provides such classifications. Viewed in this way, the MIRV in-memory tree is simply the abstract syntax tree produced by the front-end C parser [1].

```
class mirvCode {
  ...
protected:
  ...  // Dataflow API
public:
  ...
  mirvCode(void);
  virtual ~mirvCode(void);

  // Make a copy of the mirvCode (virtual copy constructor)
  virtual mirvCode *clone(bool copyDataflow = false) = 0;

  ... // Attribute API
  ... // Program structure API
  ... // Dataflow API
  ... // Transformation API
  ... // Compiler debugging/profiling API
};
```

Figure 2.6: The `mirvCode` Base Class

```
class mirvCode {
...
public:
...
  // Attribute API

  // String attribute methods
  // Get or set a string attribute
  std::string& stringAttribute(stringAttributeTag name) const;

  // Check if the mirvCode has a string attribute
  bool hasStringAttribute(stringAttributeTag name) const;

  // Reset all string attributes with the given name
  static void removeStringAttributes(stringAttributeTag name);

  ... // Similar methods for int, bool, double and node attributes

  // Convenient casting access to node attributes
  template<class A> A &getInternalNodeAttribute(nodeAttributeTag name) const;
  template<class A> void setInternalNodeAttribute(nodeAttributeTag name,
                                                  const A &attr) const;

  void removeMostMyAttributes(void);  // Remove linearizable attributes
  void removeAllMyAttributes(void);   // Remove all attributes

  // Find if a node attribute type exists anywhere in the program
  static bool hasNodeAttributeInProgram(nodeAttributeTag name);
  ...
};
```

Figure 2.7: The `mirvCode` Attribute API

```
class mirvCode {
...
public:
...
  // Program structure API

  // Compare for equality with another mirvCode (tree matching)
  virtual bool operator==(const mirvCode &rhs) const = 0;
  bool operator!=(const mirvCode &rhs) const;

  // Context methods.  These return NULL if invalid (i.e. getting a
  // parent statement on a module.
  mirvCode *getParent(void) const;

  virtual mirvModuleSymbol *getParentModule(void) const;
  ... // Similarly for function, statement, if, loop, block

  // A generic getParent (i.e. getParentOfType<mirvExpression>())
  template<class A> A *getParentOfType(void) const;

  // Get a parent that fits some criteria (type, existance of attributes, etc.)
  // P is a std-type predicate function object
  template<class P> mirvCode *getParentWithProperty(P &pred) const;

  // Find program join points.  Optionally return the block-level statements
  // containing ``this'' and ``code.''
  mirvBlockStatement *getCommonParentBlock(const mirvCode *code,
                                           const mirvStatement **t = 0,
                                           const mirvStatement **c = 0) const;

  // Check parent/child relationships
  bool isParentOf(const mirvCode *c) const;
  bool isContainedBy(const mirvCode *c) const;

  bool dominates(const mirvCode *c) const;
  bool postDominates(const mirvCode *c) const;

  bool executesAfter(const mirvCode *c) const;
  bool executesBefore(const mirvCode *c) const;

  // Transformation API
  void replaceWith(mirvCode *, bool setChanged = true);
};
```

Figure 2.8: The `mirvCode` Structure and Transformation API

Common Interfaces

Every class representing a program element in MIRV derives from the `mirvCode` class, sketched in figure 2.6. `mirvCode` provides the most common operations needed by various phases of the compiler front-end: attribute access (figure 2.7), tree walking, code structure relations and code structure transformation (figure 2.8). The `mirvCode` interface only provides tree-walking abilities to visit parent nodes because while every tree element has a parent, not every tree element has children. It is interesting to note that many important program structure analyses only require knowledge of a node's parent. For example, we often want to know whether a statement is within a loop or whether a data reference is to an r-value or l-value[8]. Providing a common parent retrieval interface allows MIRV to implement the dominator computation by checking whether parent nodes of the potentially dominating node include conditionals (loops, if-statements, etc.) and whether the subtree containing the potentially dominating node executes before the subtree containing the potentially dominated node in a common parent block.

Attributes are implemented simply using arrays of `hash_map` indexed by an *attribute tag* and hashed by the `mirvCode` object's address (the `this` pointer value). A more advanced implementation might use `boost::property_map` but the simple prototype implementation has proven easy-to-use, if slightly slow [15]. Some common attribute tags for linearizable attributes are listed in figure 2.9.

Symbol Table

The MIRV symbol table holds all of the necessary information about data objects such as their type, size, value (for constants) and so forth. All symbol classes in MIRV derive

[8]Roughly speaking, an *l-value* is a piece of data that has an address while an *r-value* does not. More roughly, *r-values* are generally unnamed expression temporaries.

1. **Integer Attributes**

   (a) `Line` – Source line number

   (b) `DynFreqCnt` – Runtime execution profile

   (c) `CallSiteTag` – Call graph annotation

2. **Double Attributes**

   (a) `ExecutionTime` – Runtime execution profile

   (b) `CumulativeExecutionTime` – Runtime execution profile

3. **String Attributes**

   (a) `Name` – Source-level symbol name

   (b) `Calls` – Call graph annotation

   (c) `CalledBy` – Call graph annotation

4. **Boolean Attributes**

   (a) `Register` – Is this data allocatable in a register?

   (b) `Unstructured` – Does this function make use of unstructured `goto`?

   (c) `Leaf` – Is this a leaf procedure (no callees)?

Figure 2.9: Common Linearizeable Attribute Tags

```
class mirvSymbol : public mirvCode {
...
public:
  mirvSymbol(unid *u);
  virtual ~mirvSymbol(void);

  virtual mirvSymbol *clone(bool copyDataflow = false) = 0;

  virtual unid *getUnid(void) const { return id; }
  void setUnid(unid *u) { id = u; }

  virtual bool operator==(const mirvCode &rhs) const;

  void setAddressTaken(bool v = true) { addrTaken = v; }
  bool addressTaken(void) { return addrTaken; }

  /// Create a canonical name for the symbol.
  virtual std::string createCanonicalName() const;

  .. // Filter API
};
```

Figure 2.10: The `mirvSymbol` Class

```
class mirvTypedSymbol : public mirvSymbol
{
...
public:
  mirvTypedSymbol(unid *u, mirvTypeSymbol *newType);
  virtual ~mirvTypedSymbol(void) {};

  virtual mirvTypedSymbol *clone(bool copyDataflow = false) = 0;

  virtual mirvTypeSymbol* getType(void) const { return type; }
  void setType(mirvTypeSymbol *t) { type = t; }

  virtual bool operator==(const mirvCode &rhs) const;

  .. // Filter API
};
```

Figure 2.11: The `mirvTypedSymbol` Class

```
class mirvTypeSymbol : public mirvSymbol {
...
public:
  mirvTypeSymbol(unid *u, unsigned int newSize);
  virtual ~mirvTypeSymbol(void);

  virtual mirvTypeSymbol *clone(bool copyDataflow = false) = 0;

  // ``int *'' for ``int,'' etc.
  void setPointerType(mirvPointerTypeSymbol *p);
  mirvPointerTypeSymbol *getPointerType(void);

  virtual bool operator==(const mirvCode &rhs) const;

  virtual unsigned int getSize(void) const;
  // Deprecated -- symbols don't change size
  void setSize(unsigned int);

  // Check if the type is structurally equivalent to this type
  // (ignore signedness, etc.)
  virtual bool isEquivalentWithoutQualifiers(const mirvTypeSymbol &t) const
    = 0;

  ... // Filter API
};
```

Figure 2.12: The `mirvTypedSymbol` Class

```
class mirvScalarTypeSymbol : public mirvTypeSymbol {
...
public:
  mirvScalarTypeSymbol(unid *id, unsigned int sz);
  virtual ~mirvScalarTypeSymbol(void);
};

class mirvAggregateTypeSymbol : public mirvTypeSymbol {
...
public:
  mirvAggregateTypeSymbol(unid *id, unsigned int sz);
  virtual ~mirvAggregateTypeSymbol(void);
};
```

Figure 2.13: MIRV Type Classification Classes

from `mirvSymbol`, shown in figure 2.10. Figure 2.11 shows the added members for typed symbols.

A symbol's type is modeled using a class derived from `mirvTypeSymbol` as presented in figure 2.12. The `setPointer()`/`getPointer()` members are a convenient way to access types related to the current type being inspected. This is particularly useful for alias analysis or any other filter that works extensively with pointer types.

Figure 2.13 presents an interesting example of inheritance as classification in MIRV. The `mirvScalarTypeSymbol` or `mirvAttributeTypeSymbol` classes neither provide new interfaces nor do they override implementations. The sole purpose of these classes is to provide a grouping of the type subclasses. Subclasses such as `mirvIntegerTypeSymbol` and `mirvFloatTypeSymbol` are categorized by (derive from) `mirvScalarTypeSymbol` while the more complex `mirvArrayTypeSymbol` and `mirvStructTypeSymbol` derive from the `mirvAggregateTypeSymbol` class. Some subclasses, such as `mirvFunctionTypeSymbol` derive from neither. Filters can use these classes to constrain the set of MIRV tree nodes upon which a particular analysis or transformation may operate. These classes are the run-time analogue to the static *concept* framework of libraries such as the C++ Standard Template Library [16]. Whereas template concept interfaces model what type parameters are allowed at compile-time, classifications model such relationships at run-time. Concept errors are flagged at compile time while classifications are checked during program execution. Mismatches can be flagged as errors or simply ignored.

Expressions

All expressions in MIRV derive from the base `mirvExpression` class shown in 2.14. Most of the members that manipulate code are used during the initial MIRV tree building

```
class mirvExpression : public mirvCode
{
private:
  int indirectionLevel;

public:
  mirvExpression(void);
  virtual ~mirvExpression(void) {};

  // Methods from mirvCode
  virtual void accept(mirvVisitor &);
  virtual bool operator==(const mirvCode &rhs) const;
  virtual mirvExpression *clone(bool copyDataflow = false) = 0;

  /// Get the type of the expression
  virtual mirvTypeSymbol* getType(void) const = 0;

  // Check if the expression is a conditional.
  // Deprecated.  Use dynamic_cast.
  inline virtual bool isCondition() { return false; }

  // Negate the expression.  The expression must be a condition.
  // This returns  a new expression.  The old expression should
  // be considered invalid.  Used only during tree building.
  mirvExpression* negateCondition();

  // Cast the expression.  This may return a new expression and
  // delete the old one.  Do not use on expressions in the tree,
  // only on newly created expressions!
  mirvExpression* castTo(mirvTypeSymbol* type);

  // Replace this expression in the MIRV tree.
  void replaceWith(mirvExpression *e, bool setChanged = true);
};
```

Figure 2.14: The mirvExpression class

```
class mirvUnaryOpExpression : public mirvExpression {
...
public:
  mirvUnaryOpExpression(mirvExpression* e);
  virtual ~mirvUnaryOpExpression(void);

  ... // mirvCode methods as in mirvExpression

  mirvCode *setOperand(mirvExpression*);
  mirvExpression* getOperand(void) const;

  ... // Dataflow API
};

class mirvBinaryOpExpression : public mirvExpression {
...
public:
  mirvBinaryOpExpression(mirvExpression* left, mirvExpression* right);
  virtual ~mirvBinaryOpExpression(void);

  ... // mirvCode methods as in mirvExpression

  mirvExpression* getLeftOperand(void) const;
  mirvCode *setLeftOperand(mirvExpression*);

  mirvExpression* getRightOperand(void) const;
  mirvCode *setRightOperand(mirvExpression*);

  ... // Dataflow API
};
```

Figure 2.15: Unary/Binary Expression Subclasses

```
// Base for direct and indirect array reference
// Arrays are first-class objects in MIRV
class mirvArrayExpression : public mirvExpression {
...
public:
  // Constructor with type of element being referenced
  mirvArrayExpression(mirvTypeSymbol *t);
  virtual ~mirvArrayExpression(void);

  ... // mirvCode methods as in mirvExpression
  ... // getType gets the element type, not the array type.

  void addIndex(mirvExpression* e);
  // i is the dimenstion index.
  mirvExpression* getIndex(unsigned int i);
  mirvCode *setIndex(indexIterator i, mirvExpression* e);

  // a la STL containers
  typedef std::list<mirvExpression*>::iterator indexIterator;
  typedef std::list<mirvExpression*>::const_iterator constIndexIterator;
  typedef std::list<mirvExpression*>::reverse_iterator reverseIndexIterator;
  typedef std::list<mirvExpression*>::const_reverse_iterator
  constReverseIndexIterator;

  indexIterator indexesBegin(void);
  constIndexIterator indexesBegin(void) const;
  reverseIndexIterator indexesRBegin(void);
  constReverseIndexIterator indexesRBegin(void) const;
  indexIterator indexesEnd(void);
  constIndexIterator indexesEnd(void) const;
  reverseIndexIterator indexesREnd(void);
  constReverseIndexIterator indexesREnd(void) const;

  bool indexesEmpty(void) const;
  int indexesSize(void) const;

... // Dataflow API
};
```

Figure 2.16: Array Expression Subclass

```
// Base for direct and indirect field reference in an aggregate type (struct).
class mirvFieldExpression : public mirvExpression {
...
public:
  mirvFieldExpression(mirvFieldSymbol* f);
  ~mirvFieldExpression();

  ... // mirvCode methods as in mirvExpression

  mirvFieldSymbol* getFieldSymbol() const;
  mirvCode *setFieldSymbol(mirvFieldSymbol* f);
};
```

Figure 2.17: Field Expression Subclass

phase. The deprecated `isCondition()` was a result of a non-functioning `dynamic_cast` in an earlier version of our build environment. Several such virtual boolean functions are scattered throughout the MIRV API.

Figures 2.15, 2.16, 2.17 and 2.18 show the immediate subclasses of `mirvExpression`. These cover most of the structural needs of expressions in MIRV (i.e. holding child operands, etc.). Note that these classes can also be used as classifications. Some concrete subclasses are listed in 2.19.

Figure 2.19 illustrates a tradeoff between lines of code and functionality. We have elected to model each possible arithmetic expression with a separate class. This makes writing filter visitors (described in section 3.2.5) slightly simpler because the C++ type system automatically allows the visitor objects to discriminate various types of expressions and ignore irrelevant structures for a particular analysis or transformation. The alternative is to store an operation tag in the `mirvUnaryOpExpression` and `mirvBinaryOpExpression` classes. This, however, would require extra work on the part of filter visitors to check and dispatch on the tag – functionality already provided by the C++ virtual function mechanism. A third option combines the best of both worlds: templated classes could subclass from `mirvUnaryOpExpression` and `mirvBinaryOpExpression`. The non-type template pa-

```
// Base for indirect and direct function calls.
class mirvFunctionExpression : public mirvExpression {
...
public:
  mirvFunctionExpression(void) {};
  virtual ~mirvFunctionExpression(void);

  ... // mirvCode methods as in mirvExpression
  // getType returns the type of the return value

  // Get the function signature
  virtual mirvFunctionTypeSymbol* getFunctionType() const = 0;

  void addParameter(mirvExpression*);
  mirvExpression* getParameter(unsigned int i);
  mirvCode *setParameter(parameterIterator i, mirvExpression* e);

  // a la STL containers
  typedef std::list<mirvExpression *> parameterList;
  typedef std::list<mirvExpression*>::iterator parameterIterator;
  typedef std::list<mirvExpression*>::const_iterator constParameterIterator;
  typedef std::list<mirvExpression*>::reverse_iterator
  reverseParameterIterator;
  typedef std::list<mirvExpression*>::const_reverse_iterator
  constReverseParameterIterator;

  parameterIterator parametersBegin(void);
  constParameterIterator parametersBegin(void) const;
  reverseParameterIterator parametersRBegin(void);
  constReverseParameterIterator parametersRBegin(void) const;
  parameterIterator parametersEnd(void);
  constParameterIterator parametersEnd(void) const;
  reverseParameterIterator parametersREnd(void);
  constReverseParameterIterator parametersREnd(void) const;

  bool parametersEmpty(void) const;
  int parametersSize(void) const;

  ... // Dataflow API
};
```

Figure 2.18: Function Call Expression Subclass

```
class mirvNotExpression : public mirvUnaryLogicalExpression { ... }
class mirvLtExpression : public mirvRelationalExpression { ... }

class mirvNegExpression : public mirvUnaryArithmeticExpression { ... }
class mirvAddExpression : public mirvBinaryArithmeticExpression { ... }
class mirvSubExpression : public mirvBinaryArithmeticExpression { ... }
```

Figure 2.19: Expression Leaf Classes

```
// Pure classification classes
class mirvUnaryArithmeticExpression : public mirvUnaryOpExpression
class mirvBinaryArithmeticExpression : public mirvBinaryOpExpression

// <, ==, etc.
class mirvRelationalExpression : public mirvBinaryOpExpression {
  // Change < to >, etc.
  virtual mirvRelationalExpression *
  createReverse(bool copyDataflow = false) = 0;
};

// Pure classification classes
// !, &&, ||, etc.
class mirvUnaryLogicalExpression : public mirvUnaryOpExpression
class mirvBinaryLogicalExpression : public mirvBinaryOpExpression
```

Figure 2.20: Expression Classifications

rameter could specify the desired operation. The template mechanism creates a unique type of each operation tag, allowing the visitors to exploit polymorphism. Note that templatizing `mirvUnaryOpExpression` and `mirvBinaryOpExpression` is suboptimal for several reasons: it would disallow the classifications built on top of them and would remove the unary/binary classification layer. Unfortunately, at the time MIRV development began, template support in C++ compilers was very poor, so the template design was not a practical one for us.

As with `mirvTypeSymbol`, several classifiers are derived from `mirvExpression`. These are listed in figure 2.20. Some of these classes are "pure" classification classes in that they do not add any new interfaces. They do override some existing virtual methods. For example, `mirvBinaryArithmeticExpression` overrides the `accept(mirvVisitor &)` method so that filter visitors may discriminate between various categories of expressions and ignore irrelevant structures. The "impure" classification `mirvRelationalExpression` adds one interface to reverse the sense of the relation. Arguably this should not even be a class member (see section 2.4 so we have grouped this class with the other classifiers.

Class Hierarchy Improvements

Overall the MIRV class hierarchy has severed us well. However, as with any large projects, many lessons have been learned along the way. Probably the most crucial one is interface reduction. The code classes have far too many interfaces. Many of these interfaces, such as node attribute manipulators, should be moved into free functions, leaving the class-proper interfaces to handle low-level access to `protected` and `private` data. Such free functions are in fact still part of the class interface, as explained by Sutter but removing them from the class itself increases encapsulation and improves understanding of the class [17].

A sore point in the tree classes is the non-uniform access to child objects. For example, block statements use an iterator interface while expression classes use `getOperand` or `getLeftOperand`/`getRightOperand` depending on whether they are unary or binary expressions. This creates an unnecessary distinction between classes that have one or two children and classes that may have more than two children. One solution is provided by the Composite design pattern [18]. Composite allows the programmer to treat leaf and non-leaf classes identically. A common iterator interface on the composite base class could make tree traversal and manipulation much simpler. Such an interface will require an abstract/virtual iterator class that is able to conduct iteration over the various types of children (statements, expressions, etc.).

Closely related to the above problem is the restriction on the number of children for certain code constructs. The assignment statement is a prime example. During the course of our work with MIRV we have occasionally found the need to insert an assignment statement referencing a machine-specific operation via the `ulit` or `blit` operators. These machine specific assignments may in fact define multiple data items, meaning that we would like to

have an assignment class capable of specifying multiple left-hand-side arguments. Our past solution has been to insert "dummy" literal assignments to specify the definitions of the additional items and force the dataflow analyzer to logically group the constructs together. A generalized interface similar to that described above would make this much easier.

We have found the classifiers extremely useful for quick pattern matches and reducing complexity in the dataflow and transformation actions. Future versions of MIRV should include more classifiers such as classes to express the mathematical properties of expressions (transitive, reflexive, etc.) and possibly additional properties that would be useful for flow visitors to match. Because multiple properties may be applicable to particular node types, representing these as mixin classes to be used with multiple inheritance seems the most appropriate design.

Finally, we quickly note that there are a number of leaf classes that could be collapsed. Currently MIRV uses distinct classes to represent, for example, binary expressions. There is a class for an add operation, a subtract operation and so forth. We have found this to be convenient because flows and actions can easily distinguish among expression types via the double-dispatch mechanism. However, maintaining this large number of classes has been troublesome. At the time of our initial design, C++ template implementations in compilers left much to be desired. Modern compilers, however, are able to handle even the most complex template specifications quite well. Many MIRV leaf classes could be combined into just a few template classes with an operator template argument.

2.5 Back-end Design

The back-end of the compiler is structured much more traditionally than the front-end and we do not go into great detail about its design in this work. This section presents

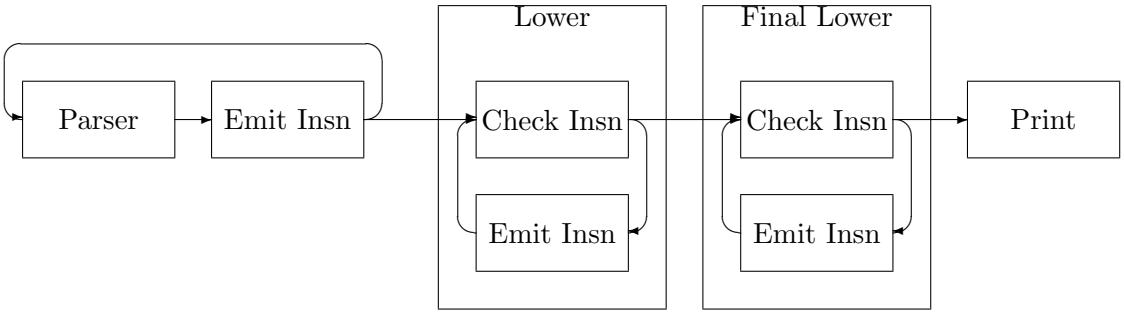


Figure 2.21: Backend Code Generation Flow

a high-level overview of the code generation process. We begin with a discussion of the "back-end process" and then proceed to describe in more detail some of the primary data structures and programmer interfaces.

2.5.1 Code Generator Flow

The primary purpose of the back-end is to convert high-level MIRV IR into low-level assembly code for the target processor. While some code transformations are performed, the set of available filters is much smaller in the back-end. Most of the transformation effort has been concentrated on the front-end.

Figure 2.21 diagrams the high-level operation of the back-end. The parser uses a syntax-directed translation scheme to generate low-level IR instructions as program elements are recognized [1]. The low-level IR is organized into the traditional list of basic blocks con-

taining quad-form pseudo-instructions. Once the low-level IR has been generated for each function. A "lowering" phase is invoked. The lowering phase ensures that the pseudo-instructions are representable on the target architecture. Typically, lowering an instruction requires breaking up complex addressing modes into individual arithmetic instructions and converting three-address forms into two-address forms for those machines that require it, such as the Intel IA32 family of processors [19]. This lowering process is recursive, meaning that instructions produced by lowering must themselves be lowered to ensure proper translation. Once lowering is completed most quads are in a form that can be mapped directly onto a processor instruction. Code that is dependent on later compilation phases, such as function prologue and epilogue sections, are kept in a higher-level form.

Following lowering, the analysis and transformation filters operate on the basic block structure. At a user-specified point in the transformation process, the register allocator converts symbolic register names into machine register names and inserts any spill code that is necessary. Transformations continue, operating on the register-allocated code. Finally, a "final-lower" phase is run on the quads. This phase expands the prologue and epilogue code and expands assembler macros. This process destroys most of the quad data structures, replacing them with literal strings that will be copied directly into the final assembly code file. After the final lowering phase it is impossible to perform further program analysis or invoke transformations that require dataflow information.

Assembler macro expansion is crucial for operation of the final back-end phases. These final phases perform any tasks that must manipulate code as close to the final executable as possible. Because the back-end does not implement linker operations it is impossible to convert symbolic label names to their final addresses. Therefore, assembler macros for global address generation cannot be expanded. Typically, such operations require at least

two instructions, one to load the upper bits of the address and one to fill in the lower bits. To compensate for this lack of expansion, the back-end attempts to determine which global addresses will require multiple instructions to compute. Some architectures, such as the PISA target, can keep certain global data in a "small data" section and address them in a single instruction using a global variable base register. The back-end determines which addresses are likely to be placed in the small data section. Other address computations generate `shim` instructions that act as place-holders to fill out the program text space. These `shim` instructions do not generate any actual assembly code. Software instruction prefetching requires that the code be in its final lowered form. Currently it is the only code manipulation performed after final lowering those operations such as code layout and other instruction cache optimizations will presumably require a similar form.

After final lowering, the IR code is printed to the assembly code file, along with the static data sections. This completes the back-end operation.

2.5.2 Data Structures

In this section we describe some of the key data structures in the back-end. The `Function` class holds the list of basic blocks. Each basic block is represented by the `bb` class. This class hold several lists of objects: a list of pseudo-instruction quads belonging to that block, a list of predecessor blocks and a list of successor blocks. The predecessor and success lists describe the program control-flow graph. In addition, a `controlFlowGraph` class exists to represent the graph in a more formalized manner, which allows common graph algorithms to operate on the program control flow graph. In addition to these lists, the `bb` class contains lists to hold dominator and post-dominator information generated by program analysis passes. In addition, several lists describe the relationship of the block

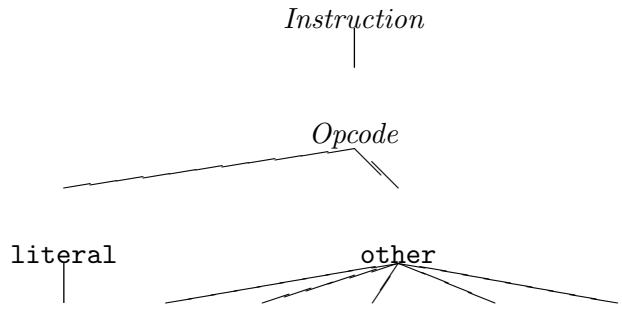


Figure 2.22: Quad Structure

to loops in the program. One such list contains references to the loops that contain the block, another the loops that the block heads and a third the loops that contain the block as their tail. These lists are used to perform various computations involving the program looping structure, such as determining how large a particular loop is in terms of number of instructions contained within it.

Surprisingly, the most complicated data structure in the back-end is the quad itself. Primarily this is due to retention of high-level addressing modes in order to facilitate efficient translation to CISC architectures. The quads are represented by the `Instruction` class, diagrammed in figure 2.22. Each quad contains an opcode describing its function and a list of *data descriptors* that describe the operands. An `Instruction` may have up to two destination operands and three source operands[9].

Each operand is represented by the `DataDescriptor` class, illustrated in figure 2.23. The data descriptor represents the location of a particular piece of data in the machine. Each object contains two discriminators that indicate the storage class of the data. The first describes the "type" of the data, `empty`, `constant` or `variable`. The second describes the "location" of the data, `reg` or `memory`. Most program data is initially placed into symbolic

[9]Meaning the quad is not actually a quad!

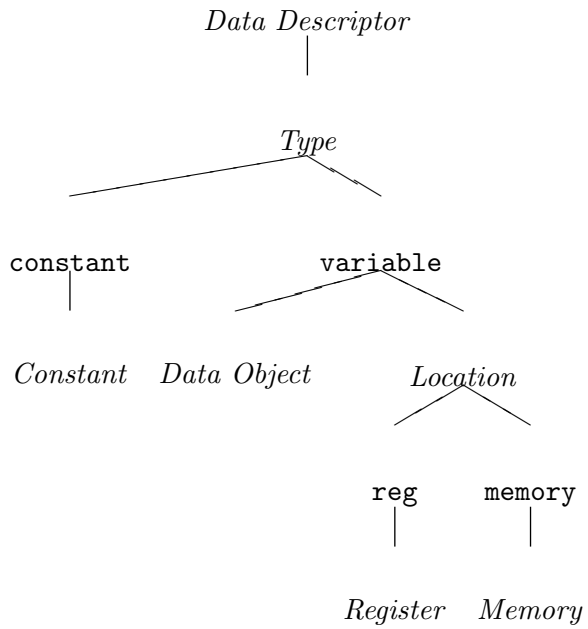


Figure 2.23: Data Descriptor Structure

registers unless it is know that the data cannot be register allocated, such as global data if global data register allocation is not being used.

Depending on the values of the discriminators the data descriptor may reference one or more sub-objects. In the case of constant data, a `Constant` object describes the type and value of the constant. Variable data may reference a `DataObject` if the data has a corresponding symbol in the program[10]. Variable data may be located in either a register or in program memory. The location discriminator describes which. In the case of enregistered data, a `RegObject` describes the symbolic or machine register holding the data while a *memory descriptor* contains the addressing information for data in core memory.

The `MemoryDescriptor` class is illustrated in figure 2.24. Each memory descriptor contains a location discriminator which indicates the addressing mode used to access the data: `nowhere`, `absolute`, `regRelative`, `memRelative`. The `absolute` mode indicates a direct label reference, as for global data. `regRelative` address indicates that the address

[10]Temporary values, for example, do not have such symbols.

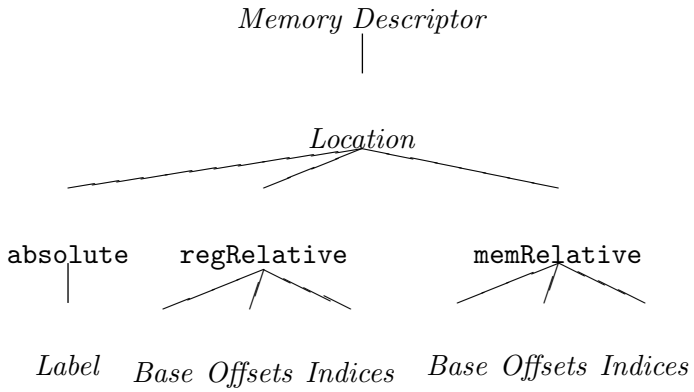


Figure 2.24: Memory Descriptor Structure

is computed relative to some base value stored in a register. The `memRelative` mode is similar, except that the base value is stored in memory.

Depending on the value of the discriminator, the memory descriptor may reference a variety of objects. In the `absolute` mode a simple label reference suffices. The `regRelative` and `memRelative` modes operate in a similar fashion. In both cases the base value is represented by a `DataDescriptor`. Thus the location discriminator is somewhat redundant in that the DataDescriptor base entirely specifies the location of the base value. However, separating out the `regRelative` and `memRelative` modes in the discriminator simplifies certain lowering checks and the computation of other information needed during various code generation phases. It also provides an opportunity for back-end consistency checking.

Relative addressing modes require that the memory descriptor keep a list of items used to compute the final address. As noted above, the back-end preserves the complex addressing modes that may be used to reference data in aggregate structures such as arrays and `struct` types. The memory descriptor may reference an arbitrary number of *offsets*, each represented by a `DataDescriptor`. These offsets are summed together with the base value to compute the final address. In addition, a list of `Index` objects may be used to perform complex addressing arithmetic to access array elements. Each `Index` contains a

`DataDescriptor` reference to the index base, usually a program variable such as a loop counter `i`. In addition, the index is multiplied by a scale factor to accommodate the type of data being referenced or to skip rows of a multidimensional array.

2.5.3 Programmer Interfaces

As in the front-end, the user may specify analysis and transformation filters on the command line via `-f<`*`filter-name`*`>`. In addition the special "filter" `-fpost` indicates the list of filters that should be run after register allocation. All other filters specified after `-fpost` will be run after register allocation has been performed.

2.6 Language Support

In this section we describe special features in MIRV to implement the supported source languages. Some of these features are merely conveniences but others are critical either for code performance or correctness.

2.6.1 C Support

The C language has little need for fancy compiler support as it is essentially a high-level assembly language [20, 21]. However, it is crucial to implement the short-circuiting operation of the logical operators efficiently. Unfortunately, the front-end currently does a poor job of translating these structures when side-effects are involved. For example if a logical expression contains a function call the front-end must break out the call into a separate statement in order to adhere to the MIRV IR syntax. Currently this is done in an inefficient manner and will be corrected in the future by efficient use of the `destAfter`/`gotoDest` structure, which was introduced long after the original parser implementation. Fortunately,

the prefix-form IR is a good structure for efficient short-circuiting code generation and this was the primary motivation for using a prefix-form tree.

Given a short circuiting expression of the form:

$$expr \rightarrow (OR \mid AND)\ expr\ expr \tag{2.1}$$

where OR and AND are the logical or (`||`) and logical and (`&&`) operators, the code generator uses an attribute grammar to easily determine the sense of the branching required. The inherited attribute specifies a *context* for the code being parsed. The context is either an *arithmetic* context for code that does not require branch generation or a *branching* context for the code that requires branches. The branching context is set when entering conditions of `ifElse` and `while` statements also upon entry to a logical operation, either of the binary form above or the unary negation form.

When in a branching context, the attribute includes additional information to describe the type of branch to emit. The *type* of branching context may be *true-fall* or *false-fall* indicating whether a branch should be taken upon a `false` or `true` result, respectively. The context also includes three target labels known as `trueLabel`, `falseLabel` and `otherLabel`. The `trueLabel` is the target of a branch in a false-fall context while the `falseLabel` is the target in a true-fall context. The `otherLabel` represents the third target necessary in some contexts: the target of the branch at the end of the `else` clause of an `ifElse` structure or the head block of a looping construct.

Operator	**Context Type**	**Actions**
`&&`	True-Fall	Generate true label, inherit false label
`||`	False-Fall	Inherit true label, generate false label
`!`	Reverse the fall sense	Swap the true and false labels

Table 2.3: Branch Context Rules

```
expr: || {
        $<ctx>$ = $<ctx>0;
        $<ctx>$->setFalseFallContext();
        $<ctx>$>->falseLabel = genLabel();
        if ($<ctx>0->isArithmeticContext()) {
          // We are the root of the logical expression subtree.
          $<ctx>$->trueLabel = genLabel();
          $<ctx>$->otherLabel = genLabel();
        }
      }
      expr {
        $<ctx>$ = $<ctx>0
        if ($<ctx>$->isArithmeticContext()) {
          // Doesn't matter what fall context we use since
          // we will not branch after this expression.
          $<ctx>$->setTrueFallContext();
          $<ctx>$->trueLabel = $<ctx>2->trueLabel;
          $<ctx>$->falseLabel = genLabel();
        }
        $<ctx>$->otherLabel = $<ctx>2->otherLabel;
        attachBasicBlock($<ctx>2->falseLabel);
      }
      expr {
        if ($<ctx>0->isarith()) {
          // Generate code to set result to 1 or 0.
        }
      }
```

Figure 2.25: Short Circuiting YACC Grammar

The various rules listed in table 2.3 govern how to set and update the context throughout the parsing process. The actions are those performed before recognition of the first non-terminal. The second non-terminal always inherits the context that the logical expression inherited because the parent expression may be another logical operation and the second non-terminal is responsible for generating the branch for that expression. The prefix-form IR ensures that these can be performed immediately upon recognition of the logical operator in a LALR parser, as shown in the YACC code of figure 2.25 [22]. Underscored rules represent epsilon actions that simply manipulate the inherited attribute passed to the non-terminal nodes. The actual generation of branches occurs in the rules for the compare operators. Their inherited context and compare operator entirely describe the type and sense of branch to emit along with the proper target label to use.

The prefix form is essential to the code of figure 2.25. Without it we would not know the type of branch to emit after parsing the first non-terminal *expr* because there is not enough lookahead to know in advance[11]. Because the logical operator appears first YACC is able to distinguish between the various types of logical and arithmetic expressions early on in the parsing process. Given an infix form, the code generator would need to examine the block of instructions produced by the first non-terminal and patch up branches to use the correct opcodes and point to the correct labels. Providing an inherited context greatly simplifies operation of the code generator.

Figure 2.26 presents an example of code generation for a short-circuiting operator. The abstract syntax tree edges are noted with the context passed to each non-terminal. Each non-terminal node is annotated with an abstract basic block representing the code generated for that non-terminal.

[11] We would be informed of this through a reduce-reduce conflict reported by YACC.

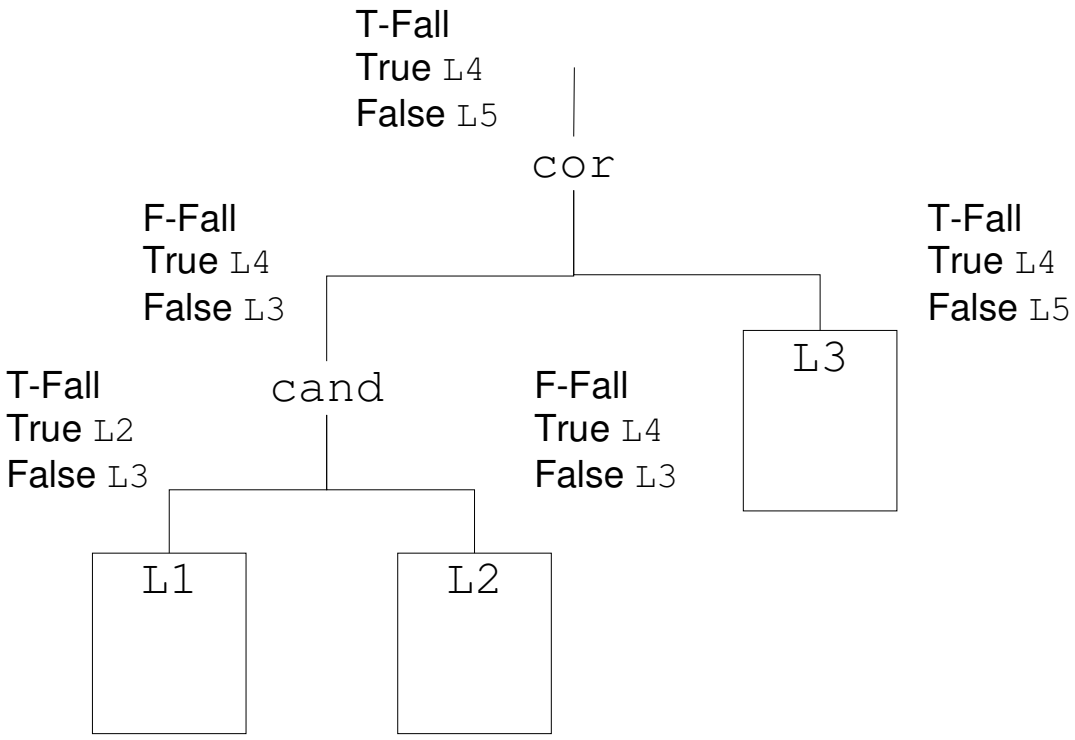


Figure 2.26: Short-Circuiting Example

2.6.2 C++ Support

The C++ language is a much higher-level language than its simpler ancestor [23, 24]. As such it requires more support from the compiler. Fortunately, our commercial front-end takes care of most of the details. In this section we describe a few additions to the commercial framework to better support real-world C++ usage.

Static Initialization

Because C++ objects have constructors that must be invoked when an object is created, global data presents a problem to the runtime system. Before the main program is executed, all of the constructors for all of the global objects must be invoked. This is traditionally done by the startup code invoked by the operating system. The startup code in turn calls

the `main` function once all of the constructors have been invoked.

MIRV uses a system similar to the original C-front `munch` utility [25]. When MIRV is invoked to link object files into a final program, the `munch` utility scans the symbol table of each object file for special well-known patterns that indicate constructors to call at startup time. It generates an additional C source file that holds a function with a special name called by the startup routines. This function walks a data structure also produced by `munch` to invoke each of the constructors in turn. This file is then compiled by MIRV and linked into the final executable.

Automatic Template Instantiation

The template facility available in C++ is a powerful mechanism to provide compile-time polymorphism. Unfortunately, due to the relatively unsophisticated link editors available, the compiler must make special efforts to present a convenient interface to the programmer. Specifically, the user would like any template code used to be instantiated automatically. Otherwise, the programmer would need to keep track of all of the argument sets used with each template and instantiate them manually through the C++ template instantiation syntax.

The EDG commercial front-end that MIRV uses includes a mechanism to support automatic template instantiation. C++ code is first "lowered" to an IR level that looks almost, but not exactly, like the IR produced by C code[12]. During this translation the front-end notes which templates are instantiated and writes special directives to an "instantiation information file" associated with each C++ source file. At link time a "prelink" utility scans these information files and re-invokes the compiler to generate the template instantiations.

[12]Structures such as block copies are preserved in the IR even though such operations are not allowed by the C language.

A new set of object files is created and this list is fed back into the final linking phase to resolve the instantiations.

Unfortunately, the supplied utilities only work out-of-the box for simplistic cases. In particular, compilation in a hierarchical source tree is problematic because the tools do not keep full paths to source and object files. A fair amount of recoding was necessary to include these paths and to invoke the MIRV compiler correctly, as the supplied tools assume that the stock EDG C++-to-C translator tool is used to compile the template instantiations.

2.7 Previous Work

Many different compiler tools have been used to conduct research in the computer architecture and wider computing fields. Unfortunately, design documents are often difficult to obtain. We describe just a few of the more common systems in use today as well as past and current systems that have influenced the development of MIRV in some way.

Probably the most well-known research compiler tool is the Stanford University Intermediate Form (SUIF) compiler [26]. Originally, the SUIF project focused on compiling for parallel machines and provided a source-to-source transformation architecture. In particular, important studies in software pipelining and alias analysis used SUIF as a foundation [27, 28]. Later work has provided extensions for native code generators and low-level optimizers [29].

The Bulldog VLIW compiler developed at Princeton University was used to explore the viability of statically-scheduled wide-issue machines [30]. Trace scheduling was an important foundation for the success of this project [31].

Hall developed a system for large-scale inter-procedural analysis and transformation [32]. In addition to the compiler proper, this system included tools for managing dependen-

cies between procedures and automatically determined which pieces of code to recompile based on previous inter-procedural analysis and transformations performed. Part of this work involved a study of inter-procedural constant propagation, procedural specialization and and procedure cloning [33]. Many of these ideas are directly applicable to the MIRV inter-procedural framework. At the moment MIRV assumes that all inter-procedural operation is performed on a whole-program linked MIRV IR. Hall's work primarily concerns a separate compilation environment though the techniques described could be used to avoid re-compilation of pieces of the linked IR file.

The IMPACT compiler tool-set introduced the concept of Explicitly Parallel Instruction Computing (EPIC) architectures, a variant of VLIW that avoids many of the shortcomings of a statically scheduled wide-issue architecture [34]. The HP Labs PlayDoh architecture is a close ancestor of IMPACT [35]. In addition to presenting the EPIC concept itself, the IMPACT team conducted studies of compiler-architecture interactions, focusing on static speculation [36, 37]. Some of this work inspired the cooperative register allocation work presented in chapters 7 and 8. The IMPACT work has found commercial application in the Intel IA-64 architecture [38] and further research expression in the Trimaran tool-set [39].

A research effort at Carnegie Mellon produced a compiler with measurable amounts of code reuse [40]. The dataflow engines of the two compilers is fundamentally different, as MIRV uses an iterative algorithm adaptor for a high-level intermediate form while the CMU compiler opts for a more traditional approach. Both compilers share the concept of a dataflow class abstraction but differ in the expression of the confluence operation. The CMU compiler uses inheritance of the dataflow traversal engine itself to provide different types of confluence while MIRV opts to express the confluence in the specific dataflow object that holds the state of the computation. The latter approach more tightly couples

the confluence operation with the dataflow representation but allows greater flexibility in specifying the design of the information architecture, as a specific confluence interface need not be assumed.

The Sharlit tool provided a mechanism for automatic generation of compiler analysis and transformation passes [41]. The Lunar compiler is a relatively new architecture proposed by Veldhuizen [42]. Initial studies are focused on new ways to describe complex compilation models for languages such as C++. Both of these systems have inspired some of the future directions for MIRV outlined in section 3.4.1.

CHAPTER 3

MIRV Dataflow Model

3.1 Introduction

This chapter explores the dataflow and transformation architecture of the MIRV compiler. Because MIRV uses a high-level tree data structure for its front-end intermediate representation, traditional analysis and transformation algorithms must be re-worked to operate within this framework. In addition, some properties of the framework allow better conceptual separation of concerns within the analysis and transformation algorithms than is traditionally presented for lower-level program representations, leading to a more cohesive yet independent collection of program manipulation passes. This in turn makes collaborative development on the compiler software much easier.

3.2 Dataflow Architecture Requirements

The heart of any optimizing compiler is the dataflow analysis and program transformation engine. Unfortunately, these are the most difficult parts of the compiler to design, implement and debug. The MIRV compiler attempts to ease the burden of designing and implementing these parts of the compiler by providing a reusable framework suitable for

```
1 a = 3;       // Def a1
2 b = 4;       // Def b1
3 if (cond) {
4    print(a); // Def a1 reaches use of a
5    a = 5;    // Def a2 - Kills def a1
6    print(a); // Def a2 reaches use of a
7 }
8 else {
9    print(a); // Def a1 reaches use of a
10 }
11 print(a, b); // Defs a1, a2 and b1 reach uses of a and b
```

Figure 3.1: Reaching Definition Example

the types of manipulations required.

Fundamentally, dataflow analysis involves two distinct actions: computation of the effects of local statements on dataflow information and propagation of that information through the program control graph with confluence operations at control join points. The MIRV dataflow framework takes care of the propagation phase, freeing the designer to concentrate on only those constructs that have a direct effect on the dataflow information.

3.2.1 An Example: Reaching Definition Analysis

Figure 3.1 presents an example of a common dataflow analysis problem: *Reaching Definition* analysis. A solution to this problem is necessary for many code transformations to ensure their correctness. The solution determines which definitions of variables can reach each use of that variable in the program. Code transformations may not violate these dependencies by, for example, moving a use above a definition that reaches it.

A simple program consisting of an if-else control structure demonstrates the flow of information though the program. Lines `1` and `2` define variables `a` and `b`. Control branches at line `3`. If `cond` is true, variable `a` is printed. The only definition of a that can reach this use is at line `1`. Variable `a` is then redefined. The print statement at line `6` has only one

definition that can reach it: the assignment at line `5`. There is no way program flow can reach this print statement without passing through this assignment. Line `9` is another use of `a`. In this case, the then-clause has been bypassed, so the original definition of `a` from line `1` reaches this use. Finally, line `11` uses both `a` and `b`.

Because the compiler does not know statically which branch of the if-else will be taken, it must be conservative in its analysis. Because `a` is not redefined along the path through the `else` branch, it must assume the definition of `a` at line `1` can reach the use at line `11`. Furthermore, the compiler must *also* assume control will flow through the `then` branch, where `a` is redefined. Thus the definition at line `5` can also reach the use at line `11`. In order to describe this conservative approach, the compiler maintains a set of reaching definitions for each possible control path through the program. At control join points, a confluence operator (normally set union or intersection, depending on the problem) is used to combine dataflow information. In the reaching definition problem, the confluence operator is set union, because the problem is to find which definitions *may* reach a given point in the program.

The example demonstrates the phases of reaching definition analysis. Each assignment statement must be examined to see where definitions are generated. This information is completely local to the assignment statement. Then, starting from the beginning of the code, the compiler examines every possible path through the code, adding definitions to the reaching set as they are encountered, and removing definitions of a variable when a new definition of that variable is seen. At control branch points, the compiler maintains separate sets of information though each path. When a merge point is encountered, these sets must be reconciled in a conservative manner. This reconciliation is achieved through the application of a *confluence operator* to the various dataflow sets.

Reaching definition analysis is a *forward* dataflow analysis, in that information flows in a forward direction through the program. There are also *backward* dataflow analysis problems where the flow begins at the end of the program and flows toward the beginning. *Live variable analysis*, used in some register allocation algorithms, is a common example of a backward dataflow analysis problem.

Reaching definition analysis is also a so-called *may* dataflow problem because its solution determines which definitions *may* (i.e. along any path) reach a given use. There are also *must* dataflow problems, such as available expression analysis. This analysis determines which expression values have been calculated but not destroyed (by, for example, modifying one or more of the expression's inputs) at various program points. In other words, it determines which values *must* reach each program point because they have been calculated along every path and not destroyed along any path leading to that point.

As we can see from the Reaching Definition example, a dataflow framework must be able to perform several different tasks, which often interact with each other:

- IR traversal
- Subtree/operator recognition (pattern matching)
- Task invocation to perform analysis on a node or subtree
- Passing of analysis results to related tree nodes

The above list combined with a high-level tree-format IR meshes nicely with a design pattern from the software engineering world: the Visitor. The Visitor pattern provides the basis for all MIRV filters in the front-end of the compiler. Given a Visitor implementation for the MIRV tree one can build filters of all different varieties.

3.2.2 The Visitor Pattern

We now provide a short tutorial on the Visitor pattern. A more formalized treatment is presented in the book by Gamma, *et al.* [18]. The Visitor pattern provides a framework for representing operations on the elements of some collection of objects. These objects are usually related to each other in some way, often through a common inheritance tree. One way to look at the Visitor pattern is that a Visitor object extends a class interface by providing new virtual functions that can be invoked on an object of that class.

Objects that are to be visited by a Visitor object must declare an interface to allow visitation. Usually this takes the form of a virtual `accept` member function. This member takes as its sole argument a pointer or reference to a Visitor object. A Visitor class declares a somewhat more complex interface, as shown in figure 3.2.

Figure 3.2 presents the `mirvVisitor` base class. This Visitor is able to visit all of the different node types in the MIRV high-level IR. A Visitor includes methods to visit all of different types of objects in the visitable collection. Each `visit` member is virtual, which allows concrete Visitors derived from `mirvVisitor` to perform different actions for each type of IR node. An example of a concrete Visitor is given in figure 3.3. This visitor is responsible for converting array references to pointer arithmetic.

Concrete Visitors often need not consider all different node types. Notice how the `mirvVisitor` class provides default implementations for all the `visit` members. By default each invocation invokes the `visit` member for the base type of the `visit` argument. This way Visitors can easily operate on entire classifications of nodes, such as binary expressions or relational expressions. The default action for the root `mirvCode` class is to do nothing, which allows concrete visitors to ignore nodes types in which it is not interested.

The key feature of a Visitor framework is known as *double-dispatch*. Double-dispatch

```
class mirvVisitor {
public:
  mirvVisitor(void);

protected:
  ...

public:
  virtual void visit(mirvCode *)              {};

  // ... visit symbols

  // ... visit statements

  // visit expressions
  virtual void visit(mirvExpression* c)
    { visit((mirvCode*) c);  };
  virtual void visit(mirvBinaryOpExpression* c)
    { visit((mirvExpression*) c); };
  virtual void visit(mirvUnaryOpExpression* c)
    { visit((mirvExpression*) c); };
  virtual void visit(mirvBinaryArithmeticExpression* c)
    { visit((mirvBinaryOpExpression*) c); };
  virtual void visit(mirvBinaryLogicalExpression* c)
    { visit((mirvBinaryOpExpression*) c); };
  virtual void visit(mirvRelationalExpression* c)
    { visit((mirvBinaryOpExpression*) c); };
  virtual void visit(mirvUnaryArithmeticExpression* c)
    { visit((mirvUnaryOpExpression*) c); };
  virtual void visit(mirvUnaryLogicalExpression* c)
    { visit((mirvUnaryOpExpression*) c); };

  // Math expressions
  virtual void visit(mirvNoneExpression *c)
    { visit((mirvExpression*) c); };
  virtual void visit(mirvAddExpression* c)
    { visit((mirvBinaryArithmeticExpression*) c); };

  // ... more math expressions
  // ... visit other expressions
  // ... more dataflow API
};
```

Figure 3.2: The `mirvVisitor` Base Class

```
class mirvArrayToPointerVisitor : public mirvActionVisitor {
public:
  mirvArrayToPointerVisitor(...);
  virtual ~mirvArrayToPointerVisitor(void) {};

  void visit(mirvArefExpression*);
  void visit(mirvArefExpression*);
  void visit(mirvAddrofExpression*);

private:
  // ...
};
```

Figure 3.3: The `mirvArrayToPointerVisitor` Class

```
void
mirvAddExpression::accept(mirvVisitor& v)
{
  v.visit(this);
}
```

Figure 3.4: The `mirvAddExpression` `accept` Member

allows the Visitor and the visited node to know nothing about the concrete type of the other at compile-time. It is implemented in the node's `accept` member, as shown in figure 3.4.

Note that both `mirvAddExpression::visit` knows nothing about the real type of the Visitor object. When a Visitor invokes `accept` on a `mirvCode` object, the virtual function mechanism ensures that the implementation for `mirvAddExpression` will be invoked if the object is in fact a `mirvAddExpression`. The accept member then invokes the virtual `visit` interface of the Visitor, passing itself as an argument. Again, the virtual function mechanism will ensure that the `visit` implementation of the concrete Visitor gets invoked. There are two virtual function calls to make the transition into the Visitor implementation, giving us double-dispatch. The `visit` member of the Visitor is essentially a new virtual function added to the `mirvAddExpression` interface, even though it is not explicitly listed in that class. It may perform any action on the `mirvAddExpression` that it wishes because it is

given a pointer to the object when it is invoked.

3.2.3 The MIRV Dataflow Framework

In the MIRV compiler, the process of flowing through the program and performing the dataflow confluence operations at the appropriate times is completely automated. The dataflow analysis designer need not be concerned about the effects of the various MIRV control constructs. The designer need only worry about four aspects of the dataflow problem:

1. When information (such as variable definitions) is generated
2. What information is removed from the dataflow set when new information is generated
3. The type of confluence operation needed
4. How to represent the dataflow information at each program point [1]

3.2.4 Dataflow Analysis Abstraction Using Attributes

The MIRV compiler uses an attribute framework to represent and manipulate dataflow information during program analysis. We now present a abstract view of this framework

Information is gathered about a program by propagating parse attributes over its structure. These attributes are referred to as *inherited attributes* when information is passed to a node before it is traversed, and *synthesized attributes* when information is returned after traversal [43]. Inherited attributes act as additional parameters given to the action code invoked upon visiting a node. Synthesized attributes represent the return values of

[1] We define *program point* as a place in the program that can use or affect the dataflow information of the analysis being performed. For example, Reaching Definition program points include assignments, uses and control points (branches and joins).

the analysis. Node attributes associate information with a node in the operator tree both during program analysis and after the analysis is completed.

By specifying what information is passed to the children of a node (member nodes) as inherited attributes and what information is passed up from a node as synthesized attributes, information can be collected about a program. The results of the attribute propagation are marked as node attributes on the operator tree.

Dataflow analysis of a MIRV program exploits the availability of high level information about the program. Both the control-flow structure as well as the expression subtrees are available in a high level, structured form. The availability of this information allows us to use a modified form of structural dataflow analysis [11, 10]. Unlike in many other compilers, the high level control-flow structure does not have to be re-synthesized from the basic-block level since it is inherent in the representation. Moreover, the basic units of analysis are not low-level quads or pseudo-instructions in basic blocks but rather operators in the MIRV tree.

In iterative dataflow analysis the structure of the program is presented as a collection of basic blocks and a control-flow graph. In order to compute dataflow information about a program, a dataflow analyzer iterates over the control graph applying equations at the basic block level until a steady state is reached.

While structural dataflow analysis also uses basic blocks as the units of computation, it assumes the existence of a hierarchical control tree. This structure can either be built up from the control-flow graph or derived from the high level program representation (as in the MIRV approach). This allows incremental updating of information as the program undergoes optimizations [44].

A variation of structural dataflow analysis is used by MIRV analysis filters because the

control tree is inherent in the MIRV program representation. The control-flow characteristics of a statement are derived from its semantic meaning. MIRV also carries the data dependence tree for expressions, so a tree based analysis method is a natural fit.

While statements in the program graph define the control-flow characteristics of the program, expressions specify the data values that are accessed. All leaf expression nodes in the tree represent a variable or constant access.

A crucial difference between MIRV dataflow and classical dataflow analysis (both iterative and structural) is that in MIRV temporary variables need not be analyzed. In the structured graph temporary values appear implicitly between expression nodes. However, temporaries have a unique characteristic that differentiates them from other variables; they are written exactly once and are read exactly once. This behavior completely defines the dataflow characteristics of the node so that these values need not be included in analysis steps, thus requiring less work to be performed.

Dataflow analysis problems can be characterized as forward or backward by the direction of traversal over the program representation. They can be further divided into two groups by classifying them as may or must problems [44]. May and must problems in a given direction usually differ only in the confluence operator that is used to combine the dataflow values that are generated during the analysis.

In MIRV, dataflow calculations are represented by the way inherited and synthesized attributes are generated during parsing. Tables 3.1 and 3.2 describes the forward and backward attribute flow behavior corresponding to statements in MIRV, respectively.

The attribute flow table denotes how attributes are propagated between operators in the language. As an example, let us examine how attributes are propagated through an `ifElse` node during backward analysis:

Operator	Child	Attribute Propagation	
		IAs for Child	**SA From Operator**
destAfter	body	$ia_{body} = ia_{destAfter}$	$sa_{destAfter} = \omega(na_{destAfter}, sa_{body})$
destBefore	body	$ia_{body} = \omega(na_{destBefore}, ia_{destBefore})$	$sa_{destBefore} = sa_{body}$
funcCall ficall	argList		$sa_{funcCall} = sa_{argList}$
gotoDest			$na_{dest} = ia_{gotoDest}$ $sa_{gotoDest} = ia_{gotoDest}$
doWhile	body condition	$ia_{body} = \omega(ia_{doWhile}, sa_{condition})$ $ia_{condition} = sa_{body}$	$sa_{doWhile} = sa_{condition}$
if	condition body	$ia_{condition} = ia_{if}$ $ia_{body} = sa_{condition}$	$sa_{if} = \omega(sa_{condition}, sa_{body})$
ifElse	condition thenBody elseBody	$ia_{condition} = ia_{ifElse}$ $ia_{ifBody} = sa_{condition}$ $ia_{condition} = sa_{condition}$	$sa_{ifElse} = \omega(sa_{ifBody}, sa_{elseBody})$
block	first node seq. node	$ia_{seqNode} = ia_{block}$ $ia_{seqNode} = sa_{previousnode}$	$sa_{block} = sa_{lastnode}$
while	condition body	$ia_{condition} = \omega(ia_{while}, sa_{body})$ $ia_{body} = sa_{condition}$	$sa_{while} = sa_{condition}$

Table 3.1: Forward Dataflow Analysis Equations

1. Inherited attribute is received for the `ifElse` node. The inherited attribute of the `ifElse` node is used as the inherited attribute of the elseBody part of the node. After setting up the inherited attribute for the member node, the filter visits it and a synthesized attribute is received from it.

2. After traversing the else part of the node, inherited attributes are set up for the ifBody member node. In this case this node also receives the same inherited attributes as the `ifElse` node.

3. After the ifBody member node is visited the condition member node is traversed. The inherited attributes for this node are produced from a combination of the synthesized attributes received from the previously visited member nodes. The results of these two nodes are combined using the confluence operator (represented by the ω symbol in the tables). This operator usually takes the intersection in must problems or the

Operator	Child	Attribute Propagation	
		IAs for Child	**SA From Operator**
destAfter	body	$na_{destAfter} = ia_{destAfter}$ $ia_{body} = ia_{destBefore}$	$sa_{destAfter} = sa_{body}$
destBefore	body	$na_{destBefore} = sa_{body}$ $ia_{body} = ia_{destBefore}$	$sa_{destBefore} = sa_{body}$
funcCall **ficall**	argList		$sa_{funcCall} = sa_{argList}$
gotoDest			$sa_{gotoDest} = \omega(ia_{gotoDest}, na_{destNode})$
doWhile	condition body	$ua_{condition} = \omega(ia_{doWhile}, sa_{body})$ $ia_{body} = sa_{condition}$	$sa_{doWhile} = sa_{body}$
if	body condition	$ia_{body} = ia_{if}$ $ia_{condition} = \omega(ia_{if}, sa_{body})$	$sa_{if} = sa_{condition}$
ifElse	elseBody thenBody condition	$ia_{elseBody} = ia_{ifElse}$ $ia_{ifBody} = ia_{ifElse}$ $ia_{condition} = \omega(sa_{ifBody}, sa_{elseBody})$	$sa_{ifElse} = sa_{condition}$
block	last node seq. node	$ia_{seqNode} = ia_{block}$ $ia_{seqNode} = sa_{previousnode}$	$sa_{block} = sa_{firstnode}$
while	condition body	$ia_{condition} = \omega(ia_{while}, sa_{body})$ $ia_{body} = sa_{condition}$	$sa_{while} = sa_{condition}$

Table 3.2: Backward Dataflow Analysis Equations

union of two sets in may problems.

4. The synthesized attribute of the `ifElse` node is simply the synthesized attribute received from the condition of the `ifElse`.

In certain cases (`destBefore`, `doWhile` and `while` nodes), there is a need to iterate over a subtree of operators multiple times. The reason for this is that there is a circular dependency between inherited and synthesized attributes of member nodes of a node. An example of this can be see in the attribute flow rules of the while operator. Here, the inherited attribute of the condition is dependent on the synthesized attribute of the body, while the inherited attribute of the body is dependent on the synthesized attribute condition.

The solution to the resolution of the circular dependency is to traverse the member nodes in the prescribed order while setting the contents of the initial synthesized attributes appropriately for the given analysis. After propagating the attributes through the member

nodes once, the received synthesized attributes can be used to compute a new inherited attributes for the nodes. This process can be repeated until the synthesized attributes reach a steady state, after which the synthesized attribute for the entire subtree can be computed. In practice, the synthesized attributes usually reach a steady state after at most two iterations (per node initiating the iteration) in MIRV programs, since the control-flow graph is reducible if there are no `gotoLabel` statements.

The attribute flow through the structured goto operators make use of the ability of associating attributes with individual operator nodes of the representation. The following procedure is used to compute a synthesized attribute for a `destAfter` node:

1. The `destAfter` node is visited before any of the associated gotoDest operators are reached (due to the structure of the MIRV representation). This allows the attribute propagator to store the current state of the inherited attributes as a node attribute at the `destAfter` node. This state corresponds to the state of the inherited attributes immediately following the body of the destination node.

2. When a `gotoDest` operator is reached, it can merge the node attribute of its destination into its attribute calculations to compute the correct synthesized attributes.

Note that the procedure is similar for the `destBefore` node with the difference that in that case the node attribute stored at the node is supplied by the synthesized attribute generated by a previous iteration through the body. Similarly to the while operator mentioned above, two passes over the body are sufficient to generate the correct synthesized attributes.

An Example: Live variable analysis

To illustrate how a real world dataflow problem can be solved using attribute propagation, we provide live variable analysis as an example. Live variable analysis seeks to determine whether there is a use of a variable on some path between a given point in the program and the exit.

Traditional live variable analysis uses the IN, OUT, DEF and USE sets to compute its result and works the following way [1]:

The first pass of the algorithm calculates the DEF and USE sets for every basic block in the graph. This is strictly a local analysis, where the DEF set corresponding to a basic block contains all variables that are defined (assigned to) in that block. The USE set contains all variables that are locally exposed uses of variables (i.e. uses of variables, whose definition comes from the outside of the basic block).

Calculation of the IN and OUT sets is performed as a succession of iterations over the control flow graph until the sets reach a steady state (The IN sets stops changing). The algorithm starts from the last basic block of the control flow graph and setting the OUT set of the last node to be the empty set. The following two equations are applied to every basic block in the graph as the blocks are traversed from back to front:

$$IN_i = (OUT_i - DEF_i) \cup USE_i \tag{3.1}$$

$$OUT_i = \bigcup_{j \in succ} IN_j \tag{3.2}$$

Live variable analysis in MIRV uses the OUT set as the inherited attributes to nodes and receives the IN set as the result of the attribute propagation through member nodes.

Operator	Child	Attribute Propagation	
		IAs for Child	SA From Operator
Unary Expression	arg		$sa_{expr} = sa_{arg}$
Binary Expression	arg 1 arg 2		$sa_{expr} = \omega(sa_{arg1}, sa_{arg2})$

Table 3.3: Live Variable Expression Equations

Since assignments in MIRV cannot be used as rvalues, the DEF and USE sets are internal information to a node and as such are not propagated.

While control-flow constructs propagate attributes, they do not usually generate them. In live variable analysis, attributes are only generated and killed by object references. Attributes are simply propagated up the expression subtrees. In other words, the synthesized attribute of the node is the combination - using the confluence operator - of the synthesized attributes if its member nodes (see Table 3.3). No inherited attributes need to be passed down to the expressions.

The use of a variable corresponds to the variable name being read by an operator. In traditional live variable analysis this would cause the accessed variable to show up in the USE set of the basic block. In MIRV analysis, the use information shows up as the synthesized attribute (IN set) of the operator that generated the use.

The equation for computing the IN set of a variable assignment makes used of the equation given in 3.1. In traditional live variable analysis writes to variables would show up in the DEF set of the unit, however in MIRV based analysis the semantics of the DEF set are conveyed by taking out the defined variable from the inherited OUT set. The resulting synthesized attribute is the union of this set and the synthesized attribute received from the source operator subtree of the variable assignment operation.

Table 3.4 illustrates the attribute flow behavior of direct object references. The OBJdest and OBJsrc nodes denote the names of the objects referenced by the operators. Describing

Operator	Child	Attribute Propagation	
		Child IAs	Operator SA
`assign`	source		$sa_{assign} = (ia_{assign} - OBJ_{dest}) \cup sa_{source}$
Obj. Reference	src		$sa_{objRef} = OBJ_{src}$

Table 3.4: Live Variable Reference Equations

the behavior of calculated object references is more difficult and dependent on how the information would be used, and the assumptions made by the compiler.

The issue is that while direct object references specify exactly which objects they reference, calculated references only specify the run-time location of the objects they reference. Depending on the aggressiveness of the compiler different assumptions can be made about the set of objects that can be referenced from such an operator. The MIRV compiler includes an alias analysis pass that attempts to restrict this set as much as possible. Another common and simple policy is to assume that all objects of the referenced type can be aliased.

3.2.5 The Attribute Flow Pattern

The previous sections describe how various analyses can be performed using attribute propagation. The presentation used an abstraction of the attribute concept to present the fundamental mechanisms to attribute-based program analysis. This section presents the concrete implementations of the attribute framework available in MIRV. MIRV filters all assume an underlying mechanism that can be used to manage the flow of attributes. Standard parsing techniques and parsing tools such as Lex/Yacc could be used as foundations for some of the filters. While Lex and Yacc can be used to efficiently drive certain filters such as our code generator, it can prove to be cumbersome if flexibility is desired in the way attributes are propagated. To provide a foundation for building filters, the Attribute

Flow pattern was designed.

Attribute Flow is based on the Visitor design pattern presented in section 3.2.2. The Visitor pattern allows an object to examine each node in a data structure (MIRV IR nodes in this context) and perform some action at each node. Normally, a Visitor object is responsible for both traversing the data structure and performing the actions. Attribute Flow decouples these two responsibilities into a traversal object and an action object. The dataflow filter designer is responsible for implementing the action object.

We have dubbed this pattern *Attribute Flow* because the traversal object has one other important responsibility: propagating dataflow information and invoking the confluence operation. Dataflow information is propagated by passing *Inherited* and *Synthesized* attributes through an Abstract Syntax Tree representation of the source program [43, 1]. In addition, we have defined a third class of attributes: *Node* attributes. Node attribute are provided to attach dataflow information to particular nodes. By using node attributes, a filter can mark the state of the dataflow at each program node and communicate dataflow information to other filters (usually transformation filters that require dataflow information to make safe code modifications).

Figure 3.5 presents a diagram of the forward Attribute Flow pattern illustrating the flow through an `if` statement. Inherited attributes are represented by the red arrows and synthesized attributes by the blue arrows. Figure 3.6 shows the interations between the different objects in Attribute Flow. There are three main objects that interact to implement the pattern. The flow object is responsible for traversing the tree and coordinating the actions of the other objects. The MIRV compiler defines both a forward and a backward flow class [2]. The attribute manager keeps a stack of attribute contexts as tree traversal

[2]There are some special-case flow classes (for printing the intermediate representation to a file, as an example) developed in the early stages of the compiler design. These will eventually be merged into the

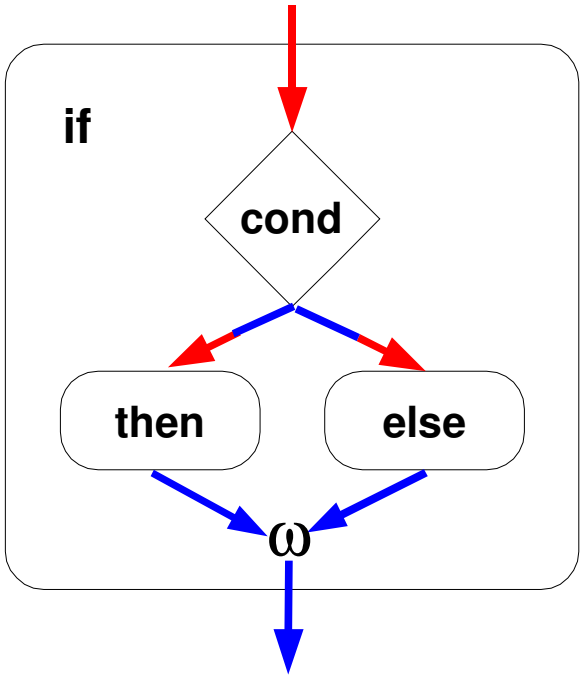


Figure 3.5: Forward Attribute Flow

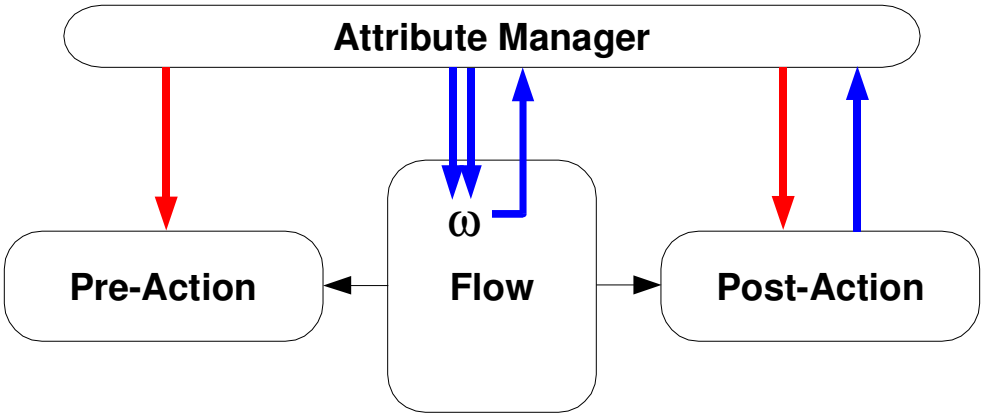


Figure 3.6: Attribute Flow Objects

progresses. It is responsible for keeping track of the inherited and synthesized attributes for a particular node and providing an interface to access these attributes. Finally, the action object encapsulates the actions to be performed at each node. Here is where dataflow information is generated and killed. After the action object has manipulated the dataflow information, the flow takes the synthesized attributes from the condition and body and performs the confluence, setting the result as the synthesized attribute for the entire `if` statement.

Decoupling the Visitor action from the flow through the program syntax tree allows the analysis or transformation designer to concentrate on only those parts of the program relevant to the problem at hand. Because the flow object handles traversing control structures and invoking the confluence operator when appropriate, the designer need not be concerned with such details. In our Reaching Definition example, the action objects operate only on node types that can either generate/kill or use definitions. Such nodes include things like function entry (to set up incoming arguments and globals), assignments, function calls (where globals and reference parameters must be assumed to be both defined and used in the worst case) and variable references (including pointer indirection). Control structures do not appear in the action class codes for this analysis.

The operation of the confluence depends entirely on how the dataflow information is represented. In the Reaching Definition analysis, definitions are represented as bits in a bit vector containing one bit for each unique definition. The confluence is then simply a bitwise `or` of the bit vectors from incoming paths at a join point. The analysis designer is responsible for implementing the confluence operator in the dataflow class itself.

The MIRV compiler currently implements the Attribute Flow pattern using a variety of

more general framework described here.

interacting objects. The include the Flow object, the Attribute Manager object and a pair of Action objects.

The Flow object encapsulates the attribute propagation behavior of a filter. It has two main roles: it determines the order in which the tree is traversed and it specifies how attributes are propagated between nodes in the tree. These two tasks are related, since attribute propagation also implies a traversal order. The Attribute Flow class has methods corresponding to all the operators in MIRV as well as to all member nodes that these operators have. The methods corresponding to member nodes of operators are used to provide a context to nodes based on the type of their parent.

The Attribute Manager object is responsible for keeping track of the attribute state and determining which attribute values are the inherited and synthesized attributes of a particular node being visited by the Flow object.

The prefix and postfix Visitor Action objects are called from the Visitor and are used to perform certain tasks before and after a node is visited. Among other things, these actions can be used to print out textual representation of the tree, perform dataflow calculations or mark parse attributes at the nodes of the tree.

MIRV Attribute Flow API

The attribute flow objects described above are modeled by a set of classes in the MIRV framework. Each of these smaller class hierarchies is essentially independent of the others in that there is no inheritance relationship between, say, flow objects and action objects. Therefore, we present each group of classes independently.

All flow classes derive from the base `mirvFlow` class. This class manages some contextual information useful to many program filters. This context tracks whether the particular

```
class mirvFlow : public mirvVisitor {
public:
  mirvFlow(...);
  virtual ~mirvFlow(void);

  // Top-level symbols
  virtual void visit(mirvPackageSymbol*);
  virtual void visit(mirvModuleSymbol*);

  // Symbols
  virtual void visit(mirvFunctionSymbol*);

  // Op Expressions
  virtual void visit(mirvNoneExpression*);
  virtual void visit(mirvBinaryOpExpression*);

  // ... More visit methods for each type of IR node
  virtual void visit(mirvUnaryOpExpression*);

  // Context information useful for dataflow analysis
  bool inAssignLHSContext(void);
  bool inAssignRHSContext(void);
  bool inUseContext(void);
  bool inDefContext(void);

protected:
  // Convenience methods to implement flows
  virtual void visitNone(mirvCode *node);
  virtual void visitSingle(mirvCode* node, mirvCode* operand);
  virtual void visitDouble(mirvCode* node, mirvCode *left, mirvCode* right);

private:
  // ...
};
```

Figure 3.7: The `mirvFlow` Base Class

```
void
mirvFlow::visit(mirvAddExpression* node)
{
  visitDouble(node, node->getLeftOperand(), node->getRightOperand());
}
```

Figure 3.8: Add Expression Flow

```
void
mirvFlow::visitDouble(mirvCode* node, mirvCode* first, mirvCode* second)
{
  flowState oldState = getFlowState();
  attributeManager.enterNode();
  beforeAction.execute(node);

  // Visit the first "child"
  attributeManager.transferInheritedDataflowAttributeDown();
  first->accept(*this);

  // Visit the second "child"
  attributeManager.transferInheritedDataflowAttributeDown();
  second->accept(*this);

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 3.9: Visiting Binary Tree Nodes

data reference being visited is in a definition or use context. Such a reference is in a definition context if, for example, it is the immediate (topmost) left-hand-side operand of an assignment statement. In addition to this context information, the `mirvFlow` class contains the implementations of tree traversal through the MIRV expression nodes. Because MIRV expressions by definition do not alter machine state, most program analyses do not care what in order these nodes are visited. If an analysis pass does depend on this order the programmer must override these expression visit methods.

Figure 3.8 shows the flow through an add expression. It uses the `visitDouble` helper which is shown in figure 3.9. As stated before the Flow object is responsible for doing IR tree traversal. This can be seen in the `visitDouble` method where each operand has its `visit` member invoked to receive a reference to the Flow object.

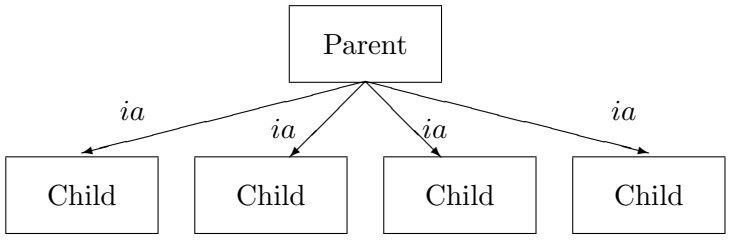
In addition to performing the actual tree traversal through invocation of `visit` members on tree nodes, the Flow object is also responsible for setting up the attribute context for each node. It does this by invoking methods on the Attribute Manager object.

```
class mirvDataflowAttributeManagerBase {
public:
  mirvDataflowAttributeManagerBase(const mirvDataflowAttribute
                                   &emptyAttributePrototype);
  virtual ~mirvDataflowAttributeManagerBase();

  virtual void enterNode(void);
  virtual void exitNode(void);
  virtual const mirvDataflowAttribute *getEmptyDataflowAttribute();

  virtual void transferInheritedDataflowAttributeDown();
  virtual void transferChildSynthesizedDataflowAttributeAcross();

  virtual const mirvDataflowAttribute *getInheritedDataflowAttribute();
  // Deprecated
  virtual void setInheritedDataflowAttribute(mirvDataflowAttribute *ds);

  virtual const mirvDataflowAttribute *getSynthesizedDataflowAttribute();
  virtual void setSynthesizedDataflowAttribute(mirvDataflowAttribute *ds);

  virtual const mirvDataflowAttribute *getChildSynthesizedDataflowAttribute();
  virtual void setChildInheritedDataflowAttribute(mirvDataflowAttribute *ds);

protected:
  // ...
private:
  // ...
};
```

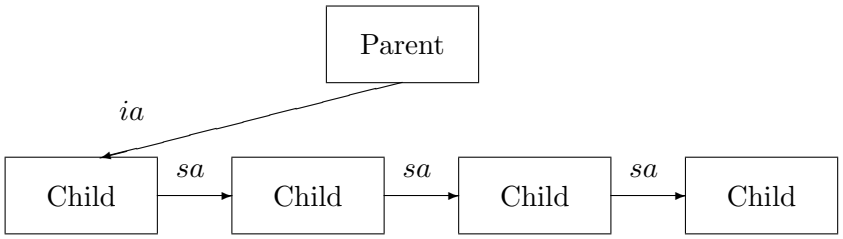
Figure 3.10: Dataflow Attribute Manager Base Class

```
template<class Attribute>
class mirvDataflowAttributeManager : public mirvDataflowAttributeManagerBase {
public:
  mirvDataflowAttributeManager(const Attribute& emptyAttribute);
  virtual ~mirvDataflowAttributeManager() {};

  virtual const Attribute &getInheritedAttribute();
  // Deprecated
  virtual void setInheritedAttribute(const Attribute& a);

  const Attribute &getSynthesizedAttribute();
  virtual void setSynthesizedAttribute(const Attribute& a);

  virtual const Attribute &getChildSynthesizedAttribute();
  virtual void setChildInheritedAttribute(const Attribute& a);
};
```

Figure 3.11: Dataflow Attribute Manager Class



(a) Down



(b) Across

Figure 3.12: Child Inherited Attribute Context Method Operation

The attribute manager object is responsible for managing the inherited and synthesized parse attributes generated during IR traversal. The `mirvDataflowAttributeManagerBase` class is shown in figure 3.10. The derived mirvDataflowAttributeManager is shown in figure 3.11. It simply adapts the base class to provide some casting convenience to recover the actual type of the attribute being managed.

The `enterNode` and `exitNode` methods are used to set up the initial attribute context for a node. They should be called at the top and bottom of a visit function, as in figure 3.9. By default a node will use its parent's inherited attribute as its own inherited attribute and `enterNode` sets up this context. Similarly, `exitNode` sets up the default synthesized attribute context. By default a node returns the synthesized attribute of the last child visited as its own synthesized attribute or if there are no children, it returns its inherited attribute as its synthesized attribute. The programmer can override

these defaults by using the other methods in the attribute manager classes. The two main methods to do this are provided by the `transferInheritedDataflowAttributeDown` and the `transferChildSynthesizedDataflowAttributeAcross` routines. The former takes a node's inherited attribute and sets its child's inherited attribute to that same node. This replicates the default behavior of `enterNode`. The latter takes the synthesized attribute from the last child visited and sets the next child's inherited attribute to that value. The operations of these methods are illustrated in figure 3.12. For the `visitDouble` method in figure 3.9 we transfer the inherited context to both child nodes because an execution order is not defined by the base `mirvFlow` class. Allowing one child to modify an attribute passed to another child does imply such an ordering so the default flow cannot do that for expressions.

The rest of the methods in the attribute manager classes allow the programmer to query the attribute context of a node and provide more fine-grained control over setting a child's inherited attribute context. For most purposes, `transferInheritedDataflowAttributeDown` and `transferChildSynthesizedDataflowAttributeAcross` are sufficient for the latter task. Both classes provide a `setInheritedDataflowAttribute` method which modifies the inherited attribute context *for the current node*. This is a deprecated interface retained for backward compatibility. Because it modifies the context of the current node it can be somewhat confusing to use.

At this point the reader may wonder why we need an attribute manager at all. The Attribute Flow pattern has many similarities to a top-down or recursive-descent parser. The C++ virtual function mechanism performs effectively the same function as the pattern matcher in a parser. If we treat Attribute Flow as a type of recursive-descent parser, it seems as though we can pass attributes the same way they are passed in such parsers

```
class mirvAction {
public:
  mirvAction() {};
  virtual ~mirvAction(void) {};

  virtual void setFlow(mirvFlow *f) { flow = f; };
  mirvFlow *getFlow(void) const { return(flow); };

  virtual void execute(mirvCode*) = 0;

  // Statements
  virtual void execute(mirvStatement* c) { execute((mirvCode*) c); };

  // ... execute members for all types of IR nodes

private:
  // ...
};
```

Figure 3.13: Action Class

– via the match routine function parameters. For Attribute Flow this means inherited attributes could be passed as an additional argument to the `visit` and `execute` functions and synthesized attributes would be return values from these functions.

This could in fact work quite well except for one limitation of the C++ language: virtual functions cannot be templates. In order to preserve the true types of the attributes we would need a template parameter to each `visit` and `execute` method describing the type of attribute being propagated. This cannot work in the current revision of the C++ language because such virtual function constructs are not allowed. We could pass attributes via references to the base `mirvDataflowAttribute` but that would require a `dynamic_cast` each time the attribute is accessed.

Separating attribute management into a distinct object solves this problem with only slight inconvenience to the programmer. Rather than having the attribute information available immediately in the parameter list, the inherited values must be received from the manager object and synthesized values must be given to the manager for proper propagation.

```
class mirvVisitorAction : public mirvAction {
private:
  mirvActionVisitor &visitor;

public:
  mirvVisitorAction(mirvActionVisitor &v) : mirvAction(), visitor(v) {};

  void execute(mirvCode* n) { n->accept(visitor); };
  // Allow use by STL-like algorithms
  void operator()(mirvCode* n) const { n->accept(visitor); };

  void setFlow(mirvFlow *f) {
    visitor.setFlow(f);
    mirvAction::setFlow(f);
  };
};
```

Figure 3.14: `mirvVisitorAction` Class

The final task of visit code such as `visitDouble` is to invoke the action code unique to each type of filter (in this case, dataflow analysis filters). Such code lives in independent *action* objects. Action objects are modeled by the base `mirvAction` class shown in figure 3.13. An action object is nothing more than a special type of Visitor, with the `visit` member replaced with `execute`[3]. Just like in the `mirvVisitor` class, each `mirvAction::execute` member by default calls the `execute` member corresponding to its object's base class. This allows analysis actions to concern themselves only with the nodes or node categories in which they are interested.

Because code like `visitDouble` of figure 3.9 factors the flow action of many different types of IR nodes into one function, it must take pointers to a common base class of all the IR nodes it operates upon. In figure 3.9 the action `execute` member is called with a `mirvCode` pointer. `mirvAction` by itself has no double-dispatch mechanism to recover the true type of the `node` parameter. Therefore, a helper object is needed to do this deduction. The `mirvVisitorAction` class presented in figure 3.14 accomplishes this task. The class

[3]This replacement was done to more easily distinguish Flow (`visit`) operations from Action (`execute`) operations.

```
class mirvActionVisitor : public mirvVisitor {
private:
  mirvFlow *flow;

public:
  mirvActionVisitor() : flow(0) {};
  virtual ~mirvActionVisitor(void) {};

  void setFlow(mirvFlow *f) { flow = f; };
  mirvFlow *getFlow(void) { return(flow); };
  const mirvFlow *getFlow(void) const { return(flow); };
};
```

Figure 3.15: `mirvActionVisitor` Class

takes a special type of Visitor in its constructor. The `mirvActionVisitor` is shown in figure 3.15. It is simply a Visitor that holds a reference to a Flow object, which allows dataflow filters to query the current context of the Flow.

When `visitDouble` invokes `mirvVisitorAction::execute`, the contained visitor is sent to `node`'s `accept` method which performs the desired double-dispatch, just as in the case of the flow objects. Thus `mirvVisitorAction` can recover the true type of `node` which was lost when `visitDouble` was invoked.

One may ask why we lost the type of `node` at all. We certainly had it when the flow's `visit` operation was called. An alternative design would in-line the code of `visitDouble` everywhere that it was called in the flow class. The code would be duplicated for may different types of IR nodes. Unfortunately, this does not lead to easy code maintenance. However, C++ templates can achieve the same effect while maintaining full type information. A prototype of the new `visitDouble` helper is shown in figure 3.16. At the time the MIRV dataflow framework was designed, our compilation environment did not have good support for class member templates. Now that such support is available we are able to implement this change in the near future. In addition, most of the current dataflow filters derive from `mirvActionVisitor` as a historical quirk because `mirvAction` did not

```
template<typename Node>
void
mirvFlow::visitDouble(Node *node, mirvCode *first, mirvCode *second)
{
  flowState oldState = getFlowState();
  attributeManager.enterNode();
  beforeAction.execute(node);

  // Visit the first "child"
  attributeManager.transferInheritedDataflowAttributeDown();
  first->accept(*this);

  // Visit the second "child"
  attributeManager.transferInheritedDataflowAttributeDown();
  second->accept(*this);

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 3.16: Templatized `visitDouble`

originally exist as a class separate from `mirvVisitor`. `mirvVisitorAction` thus exists not only as a double-dispatch bridge for code like `visitDouble` but also as an interface bridge from `mirvVisitor` to `mirvAction`. Templatizing `visitDouble`-like code and removing the existing interface bridges will completely eliminate the need for the `mirvVisitorAction` and `mirvActionVisitor` classes as well as increase compiler performance by eliminating virtual function calls and in-lining template code.

The `mirvFlow` class is used for general IR traversal. Two additional classes build upon `mirvFlow` to implement a tree traversal and attribute propagation. These two flows are used throughout the compiler to implement most of the available filters.

The `mirvForwardFlow` class implements traversal and attribute propagation common to forward dataflow problems such as Reaching Definition analysis or Available Expressions analysis. `mirvForwardFlow` provides implementations for the pure virtual statement `visit` members of `mirvFlow` and overrides the default behavior of helper functions such as

```
void
mirvForwardFlow::visitDouble(mirvCode *node,
                             mirvCode *first,
                             mirvCode *second)
{
  mirvFlow::flowState oldState = getFlowState();

  attributeManager.enterNode();
  beforeAction.execute(node);

  // Visit the first child.
  attributeManager.transferInheritedDataflowAttributeDown();
  first->accept(*this);

  // Visit the second child using first child's attributes
  attributeManager.transferChildSynthesizedDataflowAttributeAcross();
  second->accept(*this);

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 3.17: `mirvForwardFlow::visitDouble`

`visitDouble`.

The implementation of `mirvForwardFlow::visitDouble` appears in figure 3.17. The only difference from the `mirvFlow` implementation is the way attributes are propagated. Instead of coping the node's inherited attribute to both children, the second child receives the first child's synthesized attribute as its inherited attribute. For an expression node the order of evaluation does not matter since expressions by definition cannot alter machine state. However, it is important that all children of an expressions can contribute dataflow information. In Available Expressions analysis, for example, each expression node contributes a value to the dataflow propagated through the code. If the parent node's inherited attribute were sent to the second child the flow would have to remember to combine the synthesized attribute of both nodes to return the correct dataflow information. Because expressions cannot kill any dataflow information, it is more convenient to simply "pass through" the

```
void mirvForwardFlow::visit(mirvIfElseStatement* node)
{
  mirvFlow::flowState oldState = getFlowState();
  attributeManager.enterNode();
  beforeAction.execute(node);
  setFlowState(mirvFlow::normal);

  // Visit condition
  attributeManager.transferInheritedDataflowAttributeDown();
  node->getCondition()->accept(*this);

  // Save condition state so it can be passed to both the ifBody
  // and the elseBody
  mirvDataflowAttribute* condState =
  attributeManager.getChildSynthesizedDataflowAttribute()->clone();

  // Visit if body
  attributeManager.setChildInheritedDataflowAttribute(condState->clone());
  node->getIfBody()->accept(*this);

  // Save the if state
  mirvDataflowAttribute* ifState =
  attributeManager.getChildSynthesizedDataflowAttribute()->clone();

  // Visit else body
  attributeManager.setChildInheritedDataflowAttribute(condState->clone());
  node->getElseBody()->accept(*this);

  // Save the else state
  const mirvDataflowAttribute* elseState =
  attributeManager.getChildSynthesizedDataflowAttribute();

  // Merge ifState and bodyState (and pass up)
  mirvDataflowAttribute* outState = ifState->merge(elseState);
  delete ifState;
  delete condState;
  attributeManager.setSynthesizedDataflowAttribute(outState);

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 3.18: `mirvForwardFlow::visit(mirvIfElseStatement *)`

```
void mirvForwardFlow::visit(mirvWhileStatement* node)
{
  attributeManager.enterNode();

  mirvDataflowAttribute* initOut;
  const mirvDataflowAttribute* incrOut;
  mirvDataflowAttribute* condIn;
  mirvDataflowAttribute* condOut = 0;
  mirvDataflowAttribute* lastCondOut = 0;

  // Visit initialization
  initOut = attributeManager.getInheritedDataflowAttribute()->clone();
  incrOut = initOut;

  while(true) {
    // Merge initOut and incrOut
    condIn = initOut->merge(incrOut);

    // We need to set condIn as the inherited attribute for the
    // while statement.  This is just a convenient place to put
    // dataflow information so that filters can just look at the
    // while statement.
    attributeManager.setInheritedDataflowAttribute(condIn);

    mirvFlow::flowState oldState = getFlowState();
    mirvCode *oldParent = getParent();
    currentStatement = node;
    beforeAction.execute(node);
    setFlowState(mirvFlow::normal);
    setParent(node);

    // Visit condition
    // The before action may have modified condIn.
    attributeManager.transferInheritedDataflowAttributeDown();
    node->getConditionStatement()->accept(*this);

    // Get cond out.  We need to clone since it will get destroyed when
    // we visit the body.
    delete condOut;
    condOut = attributeManager.getChildSynthesizedDataflowAttribute()->clone();

    // ... more (body code visit)
```

Figure 3.19: `mirvForwardFlow::visit(mirvWhileStatement *)`, Part 1

```
    // ... visit body

    bool exit = false;
    // Check if fixed-point reached
    if (lastCondOut != 0 && (*condOut == *lastCondOut)) {
      exit = true;
    }
    if (!exit) {
      // Not at fixed point yet - save condOut
      delete lastCondOut;
      lastCondOut = condOut->clone();

      // Visit body
      attributeManager.setChildInheritedDataflowAttribute(condOut->clone());
      node->getWhileBody()->accept(*this);
      incrOut = attributeManager.getChildSynthesizedDataflowAttribute();
    }

    setFlowState(oldState);
    afterAction.execute(node);
    if (exit) {
      break;
    }
  }

  // Set synthesized attribute
  attributeManager.setSynthesizedDataflowAttribute(condOut);

  // Delete outs
  delete lastCondOut;
  delete initOut;

  attributeManager.exitNode();
}
```

Figure 3.20: `mirvForwardFlow::visit(mirvWhileStatement *)`, Part 2

first child's dataflow information though the second child.

Figures 3.18, 3.19 and 3.20 show the more complicated IR traversal provided by the forward flow. Traversing the `ifElse` node is relatively straightforward. The flow simply traverses through the condition and passes the dataflow results to each branch of the statement. After traversing each branch, the flow invokes the virtual `mirvDataflowAttribute::merge` routine to perform the confluence. This way the flow through the IR can be decoupled from the dataflow confluence, which allows the filter designer to maintain dataflow information in the most appropriate form for a particular analysis.

Traversing a loop is a bit more complicated. The loop body must be continually visited until a consistent set of dataflow information is produced. We can see from figures 3.19 and 3.20 that the flow objects perform an iterative dataflow analysis. Section 3.4.1 discusses a prototype design for a flow implementing structural dataflow analysis [11, 10]. As part of the iterative analysis, the flow must save off attribute state to be compared against later. This is because the inherited attributes "belong" to the node being visited and thus may be altered upon visiting the node. Implementation of copy-on-write semantics for dataflow attributes is a future goal. Note that the synthesized attribute of the `while` statement is the output of the condition expression. This maintains the semantics of the IR because the exit branch is defined to be taken after the condition is evaluated.

Figures 3.21, 3.22, 3.23 and 3.24 list the backward flow's `visitDouble` and two `visit` routines. Notice that `visitDouble` visits the child node in the opposite order from the forward flow. It makes no difference for expressions, but it makes all the difference for assignment statements. In the forward flow we want to make sure all uses seen in the right-hand-side of the assignment are processed before the definitions in the left-hand-side. Likewise, the backward flow needs to process definitions before uses.

```
void mirvBackwardFlow::visitDouble(mirvCode* node,
                                   mirvCode* first,
                                   mirvCode* second)
{
  mirvFlow::flowState oldState = getFlowState();

  attributeManager.enterNode();
  beforeAction.execute(node);

  // Visit the second child
  attributeManager.transferInheritedDataflowAttributeDown();
  second->accept(*this);

  // Visit the first child using second child's attributes
  attributeManager.transferInheritedDataflowAttributeDown();
  first->accept(*this);

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 3.21: `mirvBackwardFlow::visitDouble`

It is important that in backward dataflow problems the current node be operated upon *after* all children have been visited. This is most easily understood in the case of an assignment. The left-hand-side of an assignment may contain multiple data uses in the form of addressing computation. Those uses may be killed by the definition when the assignment executes. In a backward dataflow analysis the uses must be generated *after* the definition has been processed, implying that expression leaves must be traverse (operated upon) first. Backward analysis filters must put their processing code in the `afterAction` while forward analysis filters must place it in the `beforeAction`. This is a result of the duality between backward and forward dataflow problems. This duality is directly supported by the MIRV dataflow framework.

The backward flow through an `ifElse` statement is exactly the dual of the forward case. Likewise for the `while` statement. Just as the dataflow equations of tables 3.1 and 3.2 are duals, so are their realizations in the MIRV dataflow IR. Each equation in tables 3.1 and

```
void
mirvBackwardFlow::visit(mirvIfElseStatement* node)
{
  mirvFlow::flowState oldState = getFlowState();
  attributeManager.enterNode();
  beforeAction.execute(node);

  // Save the in state so it can be passed to the else
  mirvDataflowAttribute* inState =
  attributeManager.getInheritedDataflowAttribute()->clone();

  // Visit the if using the inherited state
  node->getIfBody()->accept(*this);

  // Save the if's state
  mirvDataflowAttribute* ifState =
  attributeManager.getChildSynthesizedDataflowAttribute()->clone();

  // Visit the else with the in state
  attributeManager.setChildInheritedDataflowAttribute(inState->clone());
  node->getElseBody()->accept(*this);

  // Save the else's state
  const mirvDataflowAttribute* elseState =
  attributeManager.getChildSynthesizedDataflowAttribute();

  // Merge ifState and elseState for the cond's in state
  mirvDataflowAttribute* toCondState = ifState->merge(elseState);

  // Visit condition
  attributeManager.setChildInheritedDataflowAttribute(toCondState);
  node->getCondition()->accept(*this);

  // By default, the cond's out state will be synthesized
  delete inState;
  delete ifState;

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 3.22: `mirvBackwardFlow::visit(mirvIfElseStatement *)`

```
void mirvBackwardFlow::visit(mirvWhileStatement* node)
{
  mirvFlow::flowState oldState = getFlowState();
  attributeManager.enterNode();
  beforeAction.execute(node);

  mirvDataflowAttribute* in;
  const mirvDataflowAttribute* bodyOut;
  mirvDataflowAttribute* condIn;
  const mirvDataflowAttribute* condOut;
  mirvDataflowAttribute* lastCondOut = NULL;

  // Save in state out
  in = attributeManager.getInheritedDataflowAttribute()->clone();
  bodyOut = in;

  // ... more (visit body)
```

Figure 3.23: `mirvBackwardFlow::visit(mirvWhileStatement *)`, Part 1

3.2 has a corresponding `visit` member in the forward or backward flow class, respectively.

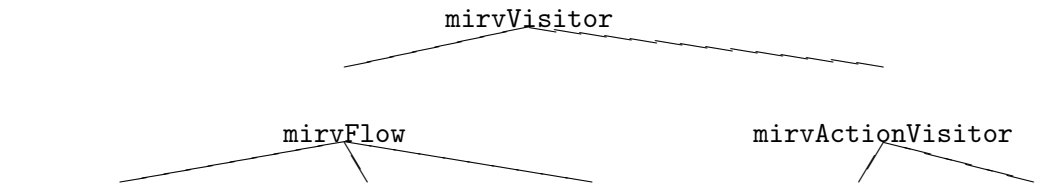
The current dataflow class hierarchy is illustrated in figure 3.25. The Forward and Backward Analysis classes can be used to perform any forward and backward dataflow analysis problem. Both of these analyses have two variants corresponding to may or must dataflow problems. However, these are not represented as derived classes in the hierarchy but rather as a confluence operator parameter to the particular analysis attribute flow object. This approach contrasts with the solution presented by Adl-Tabatabai, *et al.*, where the different confluence operators are expressed through inheritance [40].

Live Variable Analysis: Redux

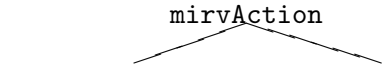
To illustrate the behavior of the use of the Attribute Flow pattern for dataflow analysis, the steps taken when computing the live variable attributes of an if node are described below and illustrated in figure 3.26. The state of the attribute stack during the analysis is presented in figure 3.27.

```
  while(true) {
    // Merge in and body-out
    condIn = bodyOut->merge(in);

    // Visit condition
    attributeManager.setChildInheritedDataflowAttribute(condIn);
    node->getConditionStatement()->accept(*this);

    // Get cond out
    condOut = attributeManager.getChildSynthesizedDataflowAttribute();

    // Check if fixed-point reached
    if (lastCondOut != NULL && (*condOut == *lastCondOut)) {
      break;
    }

    // Not at fixed point yet - save condOut
    delete lastCondOut;
    lastCondOut = condOut->clone();

    // Visit body
    attributeManager.setChildInheritedDataflowAttribute(condOut->clone());
    node->getWhileBody()->accept(*this);

    // Get body out
    bodyOut = attributeManager.getChildSynthesizedDataflowAttribute();
  }

  // By default, cond's attribute will be synthesized
  attributeManager.setSynthesizedDataflowAttribute(lastCondOut);

  // Delete out's
  delete in;

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

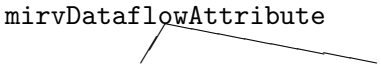
Figure 3.24: `mirvBackwardFlow::visit(mirvWhileStatement *)`, Part 2



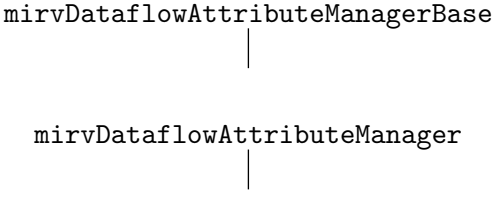
(a) Visitors



(b) Actions



(c) Dataflow Attributes



(d) Dataflow Attribute Managers

Figure 3.25: Dataflow Class Hierarchy

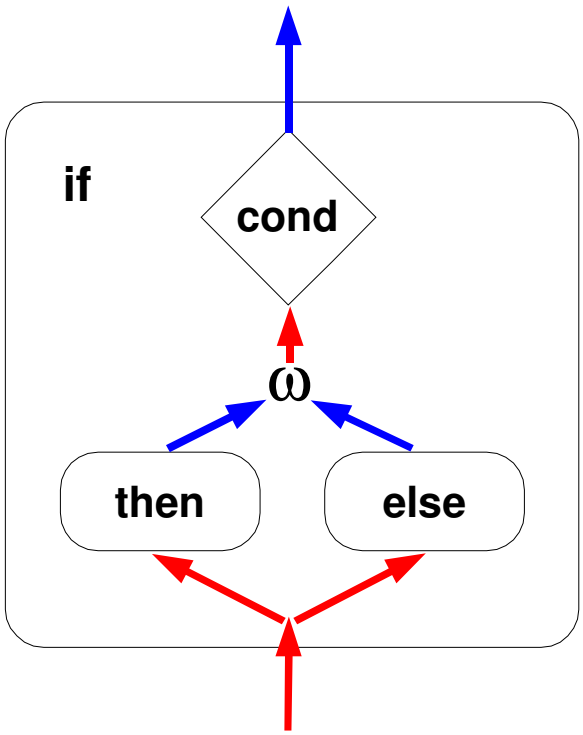


Figure 3.26: Backward Attribute Flow

1. The `ifElse` node is visited and is provided a set of inherited attributes. These attributes are copied into a node attribute context that is set up when the `enterNode` method is invoked on the Attribute Context Manager. The appropriate visitor action method is invoked on the pre-node visitor action.

2. The attribute flow of the then-body member node is invoked. The method creates a member node context and sets up the inherited attributes for the node. In this case, the inherited attributes of the `ifElse` node are simply propagated to the body. The attribute flow method invokes the appropriate visitor method on the Visitor and the attribute flow process continues with the visitation of the body.

3. The attribute flow of the else-body member node is invoked. As in the flow through the then-body, the inherited attributes of the `ifElse` node are passed to the else-body.

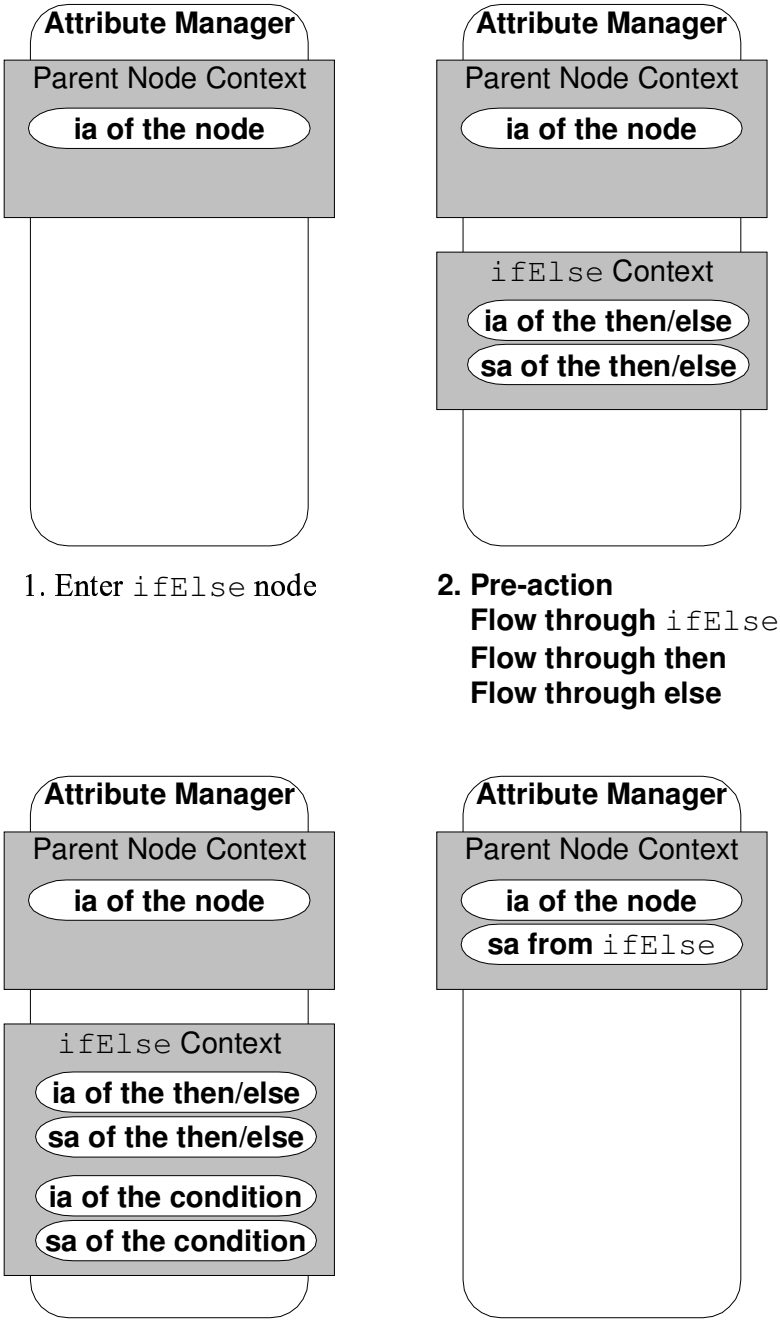


Figure 3.27: Attribute Stack State for Backward Flow

4. The attribute flow of the if-condition member node is invoked. A member node context is created in the Attribute Context Manager and the appropriate inherited attributes are passed to the node. In this case, the inherited attribute consists of the combination of the synthesized attribute of the if-body and the else-body. The appropriate visitor method is invoked for the if-condition on the Visitor object.

5. After the attribute flow methods are completed, execution continues in the `ifElse` flow. The post-node action is invoked, which saves the last member node's synthesized attribute as the IN attribute of the if-node. The Attribute Context Manager frees up the space associated with the node's member contexts and propagates the resulting synthesized attribute to the enclosing member context's synthesized attribute slot.

We now present the concrete implementation of Live Variable analysis in the MIRV compiler. For the sake of brevity we only show dataflow manipulation for the `ifElse` node as described above.

Given the MIRV Attribute Flow framework, the dataflow analysis designer must complete just a few tasks to create a functioning analysis filter. These tasks are given assuming a "standard" dataflow analysis problem that can be expressed within the *Forward*, *Backward*, *May*, *Must* framework. More complex analyses can certainly be written but require more work on the part of the filter designer. There are four main tasks the filter designer must perform:

1. Determine at which point in the $\langle\langle Forward, Backward\rangle, \langle May, Must\rangle\rangle$ matrix the analysis lives

2. Design the dataflow information representation (the attribute class)

3. Determine which program points are relevant to the analysis and implement action methods for them

4. Package these objects into the Attribute Flow framework and invoke the analysis

We now step through each of these tasks for Live Variable analysis.

The first task is very straightforward for well-known analyses. Live Variable analysis determines which variables are potentially accessed after a particular program point. The word "potentially" is an immediate indication that this analysis is a *May* dataflow problem. Determining the flow direction is only slightly more complex. Because the description of the problem tells us that we are trying to determine which variables are potentially accessed *after* a particular program point, the implication is that we need some idea of "future" knowledge about the program's data access patterns. The only way to acquire this knowledge is to look at later program points before earlier ones. This clearly indicates the need for a *Backward* flow.

To design the dataflow attribute class we first need to understand what the dataflow problem itself should provide an an answer. Because we are concerned with which *variables* are live at a program point, it is natural to assume that we somehow must represent variables in our dataflow attribute. Conveniently, MIRV already provides such a representation in `mirvVariableSymbol`. Note that we do not include other C data items such as dynamically allocated memory. This is because our use of Live Variable analysis is primarily to determine potential register pressure in the machine code that will be generated. Since dynamically allocated data cannot generally be placed in a register[4] we do not concern ourselves with representing such data[5]. A similar argument holds for aggregate field and array element

[4]But see chapters 7 and 8.

[5]Such data items must be represented for filters such as Reaching Definition analysis because other transformations depend on the production of conservative static data dependence information.

```
class mirvLiveVariableDataflowAttribute : public mirvDataflowAttribute
{
public:
  typedef set<mirvVariableSymbol*> liveSet;
  typedef set<mirvVariableSymbol*>::iterator liveSetIterator;

  mirvLiveVariableDataflowAttribute() : mirvDataflowAttribute() {};
  virtual ~mirvLiveVariableDataflowAttribute() {};

  void addLive(mirvVariableSymbol* v);
  void eraseLive(mirvVariableSymbol* v);
  const liveSet& getLiveSet();

  virtual mirvDataflowAttribute* clone() const;
  virtual bool operator==(const mirvDataflowAttribute& da) const;
  virtual mirvDataflowAttribute* merge(const mirvDataflowAttribute* da) const;

private:
  liveSet lives;
};
```

Figure 3.28: Live Variable Dataflow Attribute

data.

Given this definition of *variable* we design a class to hold this information. This class, `mirvLiveVariableDataflowAttribute`, is presented in figure 3.28. As noted above we represent variables with the `mirvVariableSymbol` class. We keep pointers to these objects because they already live in the MIRV symbol manager. We choose to use the `std::set` template out of the standard C++ library for convenience. Other representations such as bit vectors are possible. Our priorities place clarity over compilation speed unless a filter is determined to have a heavy impact on compilation time[6].

In addition to defining the member data of the class, we provide some methods to manipulate that data, such as the addition of members to the set. We also provide three methods needed by the Attribute Flow classes: `clone`, `operator==` and `merge`. The `clone` method is used to make a copy of the dataflow information. Because the flow classes keep references to the base `mirvDataflowAttribute` class it must have some way of copying

[6]Reaching Definition analysis is one such filter and we use bit-vectors there.

```
void mirvLiveVariableDataflowAttribute::addLive(mirvVariableSymbol* v)
{
  lives.insert(v);
}
```

Figure 3.29: Live Variable `addLive` Implementation

```
void mirvLiveVariableDataflowAttribute::eraseLive(mirvVariableSymbol* v)
{
  lives.erase(v);
}
```

Figure 3.30: Live Variable `eraseLive` Implementation

through the "real" type of the attribute objects. The virtual `clone` method provides that. The equality operator is used to determine when a fixed point in the dataflow analysis has been reached. Finally, the `merge` method implements the confluence operation for our Live Variable dataflow attribute. This also has the advantage of simplifying the presentation below.

Implementations of member methods are shown in figures 3.29, 3.30, 3.31, 3.32, 3.33 and 3.34. The `getLiveSet` member is a bit suspect because it breaks encapsulation by exposing the underlying data structure to the programmer. A better interface would provide iteration methods as in the standard C++ library containers. The `merge` method uses a loop to perform insertion of new members into the current dataflow attribute. Conceptually we are computing the union of two sets of variables, implementing *May* semantics for the confluence operation. At the time this analysis filter was written, standard library algorithms were not fully available to use. The standard algorithm `std::set_union` would be a more appropriate

```
const mirvLiveVariableDataflowAttribute::liveSet&
mirvLiveVariableDataflowAttribute::getLiveSet()
{
  return lives;
}
```

Figure 3.31: Live Variable `getLiveSet` Implementation

```
mirvDataflowAttribute*
mirvLiveVariableDataflowAttribute::clone() const
{
  return new mirvLiveVariableDataflowAttribute(*this);
}
```

Figure 3.32: Live Variable `clone` Implementation

```
bool mirvLiveVariableDataflowAttribute::
operator==(const mirvDataflowAttribute& da) const
{
  const mirvLiveVariableDataflowAttribute* rhs =
    dynamic_cast<const mirvLiveVariableDataflowAttribute*>(&da);

  if (rhs != NULL)
    return (lives == rhs->lives);

  return false;
}
```

Figure 3.33: Live Variable `operator==` Implementation

```
mirvDataflowAttribute* mirvLiveVariableDataflowAttribute::
merge(const mirvDataflowAttribute* da) const
{
  const mirvLiveVariableDataflowAttribute* rhs =
    dynamic_cast<const mirvLiveVariableDataflowAttribute*>(da);

  assert(rhs != NULL);

  mirvLiveVariableDataflowAttribute* newDA =
    new mirvLiveVariableDataflowAttribute(*this);

  // The new live set is the union of the two old ones
  newDA->lives = this->lives;
  for (liveSetIterator i = rhs->lives.begin(); i != rhs->lives.end(); ++i)
    newDA->lives.insert(*i);

  return newDA;
}
```

Figure 3.34: Live Variable `merge` Implementation

choice here.

We have accomplished step two of the tasks necessary to implement a Live Variable analysis filter. Step three requires us to determine the relevant program points for the analysis and to implement action methods for them. Because we are computing which variables are live (potentially accessed) the `vref` expression is an obvious program point of concern. Each of these expressions will add an element to the live variable set. Since the definition of a variable ends the liveness range for a particular value, definition points are also of interest to us. These include assignments, function calls and any other points where data may be defined. For this analysis we simplify things by assuming that definitions can only occur at assignment statements. For our use of the live variable information, this is just fine. We use the information only to estimate register pressure. Thus we do not need full correctness. Furthermore, since global variables and variables that have their addresses taken cannot live in traditional register files we ignore them. This leaves only local variables that do not have their addresses taken. Such variables can only be defined through assignment statements in the MIRV IR.

In addition to operating at the program points that generate or kill liveness information, we would like the filter to annotate each statement with the variables live into the statement and the variable live out of the statement. These node attributes correspond to the IN and OUT sets of traditional basic-block level iterative dataflow analysis. Thus every statement in the IR is of interest to our filter. The OUT set is available upon entry to a node in the backward flow, implying that the OUT node attribute will be set by the flow pre-action. The IN set is available only after visiting all children of the statement so its node attribute must be set in the post-action. The node attributes set are modeled by the class in figure 3.35. This class is extremely straightforward so we do not discuss it further.

```
class mirvLiveVariableNodeAttribute : public mirvNodeAttribute
{
public:
  typedef set<mirvVariableSymbol*> liveSet;
  typedef set<mirvVariableSymbol*>::iterator liveSetIterator;

  mirvLiveVariableNodeAttribute(void) {};
  mirvLiveVariableNodeAttribute(const liveSet&);
  virtual ~mirvLiveVariableNodeAttribute() {};

  // Get the set of variables that are live before the statement is executed
  static const liveSet& getLiveIn(mirvStatement* s);

  // Get the set of variables that are live after the statement is executed
  static const liveSet& getLiveOut(mirvStatement* s);

  // Get the set of variables in *this* attribute (rather than from a
  // mirv statement)
  const liveSet& getLiveSet();

  mirvLiveVariableNodeAttribute *clone(void) const {
    return(new mirvLiveVariableNodeAttribute(*this));
  };

private:
  // ...
};
```

Figure 3.35: Live Variable Node Attribute

```
class mirvLiveVariablePreVisitor : public mirvActionVisitor {
private:
  typedef (mirvDataflowAttributeManager<mirvLiveVariableDataflowAttribute>
    attributeManagerType;

public:
  mirvLiveVariablePreVisitor(attributeManagerType &a);
  virtual ~mirvLiveVariablePreVisitor() {};

  virtual void visit(mirvFunctionSymbol*);
  virtual void visit(mirvStatement*);

private:
  attributeManagerType &attrMan;
};
```

Figure 3.36: Live Variable Pre-Action

```
class mirvLiveVariablePostVisitor : public mirvActionVisitor {
private:
  typedef (mirvDataflowAttributeManager<mirvLiveVariableDataflowAttribute>
    attributeManagerType;

public:
  mirvLiveVariablePostVisitor(attributeManagerType &a);
  virtual ~mirvLiveVariablePostVisitor() {};

  virtual void visit(mirvStatement*);
  virtual void visit(mirvVrefExpression*);

private:
  attributeManagerType &attrMan;
};
```

Figure 3.37: Live Variable Post-Action

```
void mirvLiveVariableVisitor::visit(mirvFunctionSymbol* fs)
{
  // Create a new inherited attribute as we enter each function.
  mirvLiveVariableDataflowAttribute attr;
  attrMan.setInheritedAttribute(attr);
}
```

Figure 3.38: Live Variable Pre-Visitor Function Visit

The pre- and post-action class interfaces are presented in figures 3.36 and 3.37. As noted in section 3.2.5 the older dataflow filters in MIRV were designed before the advent of `mirvAction`. Thus these classes are based upon `mirvVisitor` and require the `mirvActionVisitor` and `mirvVisitorAction` bridge classes to operate within the Attribute Flow framework.

Implementation of the pre-visitor is shown in figures 3.38 and 3.39. Because this is a backward dataflow analysis the pre-actions will tend to be simpler than the post-actions. In this case we simply need to set the initial conditions upon function entry. We begin with no variables being live at exit from the function. At every statement we set the OUT attribute to be the attribute inherited by the statement. Because of the way Attribute Flow works, this inherited attribute will be the result of the flow through the statement following the

```
void mirvLiveVariableVisitor::visit(mirvStatement* s)
{
  // Get rid of stale information
  mirvNodeAttribute* currentAttribute = s->nodeAttribute(MirvLiveOut);
  delete currentAttribute;

  // Set this statement's live variables
  mirvLiveVariableDataflowAttribute attr = attrMan.getSynthesizedAttribute();

  mirvLiveVariableNodeAttribute* lvn =
    new mirvLiveVariableNodeAttribute(attr.getLiveSet());

  s->nodeAttribute(MirvLiveOut) = lvn;
}
```

Figure 3.39: Live Variable Pre-Visitor Statement Visit

```
void mirvLiveVariablePostVisitor::visit(mirvStatement* s)
{
  // Clear stale state
  mirvNodeAttribute* currentAttribute = s->nodeAttribute(MirvLiveIn);
  if (currentAttribute != NULL) delete currentAttribute;

  // Set this statement's live variables
  mirvLiveVariableDataflowAttribute attr = attrMan.getSynthesizedAttribute();
  mirvLiveVariableNodeAttribute* lvn =
    new mirvLiveVariableNodeAttribute(attr.getLiveSet());
  s->nodeAttribute(MirvLiveIn) = lvn;
}
```

Figure 3.40: Live Variable Post-Visitor Statement Visit

current one. This attribute is exactly equal to the IN attribute of that statement as we will see shortly.

Figures 3.40 and 3.41 show the pre-visitor actions for Live Variable analysis. Again, the statement visitor simply sets the node attribute for the statement. Because this action occurs after the statement has been processed we set the IN node attribute of the statement to the attribute synthesized by this statement. Recall that the pre-visitor sets the OUT attribute of statements. That OUT attribute is exactly equal to the IN attribute of the following statement because the flow through sequential tree nodes specifies a copy of IN to OUT attributes (see table 3.2).

```
void
mirvLiveVariableVisitor::visit(mirvVrefExpression* v)
{
  // Add this variable to the live variable set if it is a use
  mirvLiveVariableDataflowAttribute attr = attrMan.getInheritedAttribute();

  // Only update live variable info for local variables (globals can't
  // be allocated to registers, so we don't care about them)
  if (v->getVariableSymbol()->isLocal()) {
    if (getFlow()->inUseContext()) {
      attr.addLive(v->getVariableSymbol());
    }
    else if (getFlow()->inDefContext()) {
      attr.eraseLive(v->getVariableSymbol());
    }
  }

  attrMan.setSynthesizedAttribute(attr);
}
```

Figure 3.41: Live Variable Post-Visitor Reference Visit

The action for handing variable references does most of the Live Variable analysis grunt-work. Fortunately, the flow object provides some useful information to reduce the work of this action. Recall that the flow keeps track of whether an expression is is a Definition or Use context. We use that to our advantage here. Variable references can either generate or kill liveness information. If the reference is in a Definition context (the immediate right-hand-side of an assignment in this case) it will kill the corresponding liveness information for that variable. If in a use context the action will add the variable to the current live set. Finally, the resulting modified attribute is sent up the tree as a synthesized attribute.

The final task in our Live Variable filter construction is the packaging of the objects we just designed into a flow object. In the MIRV compiler each filter is represented by a `plugin` class. Figure 3.42 shows this class for Live Variable analysis. A plugin is the user interface to the analysis class. In addition to actually invoking the analysis flow, the plugin registers various user-configurable command-line options for the filter.

```
class mirvLiveVariablePlugin : public mirvPlugin {
#ifdef STATIC_BUILD
  static mirvPluginRegistrant registrant;
#endif

public:
  mirvLiveVariablePlugin(void) : mirvPlugin("liveVariable") {}
  virtual ~mirvLiveVariablePlugin(void) {};

  void registerOptions(optionDatabase *odb);

  filterRunTime  getRunTime(void)  { return anyTime; }
  filterRunLevel getRunLevel(void) { return anyLevel; }

  void activate(mirvFunctionSymbol *function);
};
```

Figure 3.42: Live Variable Plugin

The `getRunTime` and `getRunLevel` methods allow the filter to specify when it should or should not be invoked. Live variable analysis can run at any time (before or after and during "main" optimizations) but only at function scope. Some filters can be run at module/whole-program scope.

The meat of the plugin is in the `activate` method, shown in figure 3.43. Before doing anything else the method checks to see if Live Variable analysis needs to be run. Each function has the ability to keep track of which dataflow information is current. If an analysis has previously been performed and no transformations have disturbed the information[7] then there is no need to run the analysis again.

If the analysis needs to be performed, the filter records this so the compiler can print out some helpful statistics later. It then begins to construct a set of visitor and flow objects to perform the analysis. The filter creates an attribute manager and the pre- and post-actions (including the necessary `mirvVisitorAction` glue) to pass to the backward flow object. The final step is the invocation of the function's `accept` method with the flow object. In

[7]Some transformation filters explicitly manage dataflow information across transformations to reduce the amount of re-computation.

```
void mirvLiveVariablePlugin::activate(mirvFunctionSymbol *function)
{
  if (function->getNAUpToDate(LiveVariable)) {
    filterStat(functionLevel, partialRun, function);
    return;
  }

  filterStat(functionLevel, completeRun, function);

  mirvLiveVariableDataflowAttribute emptyAttribute;
  mirvDataflowAttributeManager<mirvLiveVariableDataflowAttribute>
  attribMgr(emptyAttribute);

  mirvLiveVariablePreVisitor lvpreVisitor(attribMgr);
  mirvVisitorAction lvpreAction(lvpreVisitor);

  mirvLiveVariablePostVisitor lvpostVisitor(attribMgr);
  mirvVisitorAction lvpostAction(lvpostVisitor);

  mirvBackwardFlow flow(lvpreAction, lvpostAction, attribMgr);

  function->accept(flow);

  mirvNullAction nullAction;
  mirvNullDataflowAttributeManager nullMgr;
  maxLiveVisitor maxVisitor;
  mirvVisitorAction maxAction(maxVisitor);
  mirvBackwardFlow maxFlow(nullAction, maxAction, nullMgr);
  function->accept(maxFlow);

  function->setNAUpToDate(LiveVariable);
}
```

Figure 3.43: Live Variable `activate` Method

addition to performing the "classical" Live Variable analysis, the filter uses the results to estimate the maximum number of registers live across each function call and within each function. This helps guide other filters such as function in-lining make better decisions about potentially harmful transformations. Once these tasks are accomplished the filter informs the function that its Live Variable dataflow information is up-to-date.

In this section we have described the MIRV implementation of attribute-based dataflow analysis in great detail. We presented the foundation Attribute Flow design pattern that is used throughout the compiler to perform dataflow analysis. We then described the concrete implementation of this pattern available in the MIRV dataflow API. Finally, the Live Variable analysis filter showcased the use of the design pattern and MIRV API. With minor details omitted or altered[8], this is a verbatim copy of the analysis currently in the MIRV compiler.

Alias Analysis in MIRV

To correctly compute reaching definition and live variable information, the compiler must perform some sort of alias analysis to determine the set of objects that may be affected by a particular name reference. MIRV does flow-insensitive, intra-procedural alias analysis. An analysis is *flow-insensitive* if the dataflow information at every program point is the same. In other words, the effects of the entire function are summarized at each program point. Our alias analysis is based on the presentation in Muchnick's book, where he also notes that flow-insensitivity is not usually problematic [44].

During alias analysis, each object referenced by the program is given a name. For example, variable `x` is given the name x. Expression `*p` is given the name $*p$. The alias

[8] Various object constructors, node attribute implementation, debug code, etc.

computation computes a set for each identifier listing other identifiers it may point to. The name p will point to anything assigned to pointer variable `p`. The name $*p$ will point to anything pointed to by members of p's point-to set. In addition, special names are used to summarize some information. The name *allGlobal* represents all global objects[9] in the program. The name *allAliasedGlobal* represents those global objects that have their address taken. *allAliased* refers to all objects that have had their address taken. *anon* refers to anonymous, or dynamic, memory. *temp* is a special object that represents the results of arithmetic. Normally general arithmetic cannot be used as a pointer, but the compiler must handle those cases where it is. *temp* is used to identify those cases[10]. The name *all* refers to every object in the program. Most names are typed, so that names of pointers may only point to names of their base type. Occasionally, a summary name such as *allAliased* may be untyped, in which case it refers to all alias objects, not just those of a specific type. *anon* and *all* are always untyped.

Throughout this discussion, data object names will be presented in *italics* while source-level names will be written in a `fixed-width` font.

Figures 3.44, 3.45 and 3.46 list the rules used for alias analysis in MIRV. The flow-insensitive nature of the computation is apparent in rule 3.2.5, as the point-to set of the right-hand-side is added to that of the left-hand-side rather than replacing it. Inter-procedural alias analysis attempts to reduce the effects of rules 3.2.5, 3.2.5, 3.2.5 and 3.2.5. Inter-procedural side-effect analysis attempts to reduce the effects of rule 3.2.5. Reaching definition analysis will assume everything in the *Ref* set is both used and modified by the callee. Inter-procedural side-effect analysis attempts to look at the function and find out what it really uses. The Binding Multi-graph maps the call parameters onto the formal

[9]Each field in a `struct` variable is treated as a separate object.

[10]The name *temp* was chosen to represent the storage location for intermediate arithmetic results.

Function Initialization:

- Make `allGlobal` of every pointer type point to `allAliasedGlobal` of the pointer base type and `anon`.
- Make every incoming pointer parameter point to `allAliasedGlobal` of the pointer base type and `anon`.
- Make `all` point to `all`.
- Make `anon` point to `anon`.
- Make `temp` point to `allAliased`.

Figure 3.44: MIRV Alias Analysis Initialization Rules

parameters of the callee, allowing the compiler to perform this analysis. MIRV implements only a simplified intra-procedural side-effect analysis in that it only reduces the number of global variables assumed to be modified "directly" by the function. Any globals whose addresses are passed through a call will be assumed to be both used and modified. Likewise, any dereference of a pointer parameter in the function body will be assumed to be a reference to any global variable.

3.3 Whole-Program Analysis and Transformation

In this section we briefly describe the mechanisms available in MIRV to support whole-program analysis and transformation. Such analyses operate on the same intermediate representation discussed in chapter 2. However, in the case of whole-program operation, the IR is a *linked MIRV* representation of the program.

MIRV can operate as an IR linker by specifying the `--link` option on the command line

- *Address-Of*: Make `&x` point to `x`.
- *Cast*:
  - *Pointer-to-Pointer* – Make cast point to the point-to set of its operand.
  - *Non-Pointer-to-Pointer* – Make cast point to `temp`.
  - *Call to malloc or calloc* – Make case point to `anon`.
- *Unary Arithmetic*: Make the result point to `temp`.
- *Binary Arithmetic*:
  - If either operand is `all`, the result is given the name `all`.
  - If either operand is `allAliased`, the result is given the name `allAliased`.
  - If either operand is a pointer, the result is given the name of that pointer.
  - Otherwise the result is given the name `temp`.
  - If this is pointer arithmetic:
    * If one or the other operand is named `temp`, ignore, as we are performing proper pointer arithmetic.
    * If both operands are names of pointers, name the result as `temp`, which points to `allAliased` from the initialization step.
    * Otherwise make the result point to the set of items pointed to by the (one) pointer operand.

Figure 3.45: MIRV Alias Analysis Expression Rules

- *Function Call*:
  - For every pointer passed to the callee, calculate the set of local names reachable from that pointer (i.e. perform the transitive closure on the point-to set for the argument). Call this set *Ref*.
  - For any pointers contained in *Ref*, add `allAliasedGlobal` and type-compatible objects in *Ref* to its point-to set.
  - Make `allGlobal` of all pointer types point to anything in *Ref*, and also `anon` (we are resetting the initial conditions for global pointers).
  - If there is a pointer return value, make it point to type-compatible names in *Ref*, or to `anon` if this is a call to `malloc` or `calloc`.
- *Assignment*: Add everything in the point-to set of the right-hand-side to the point-to set of the left-hand-side.

Figure 3.46: MIRV Alias Analysis Statement Rules

```
struct {
  int a;
  int b;
} global;
```

Figure 3.47: An Anonymous `struct` Type

and providing a set of MIRV IR files on which to operate. The compiler will read in each IR file and perform a symbol renaming to match up corresponding symbols from each file. This has a number of consequences that we outline here.

When performing the link, global names must be maintained. That is, any names necessary for correct binary linking, such as procedure names, global variable names and other such symbols must not be renamed in any way. On the other hand, names local to a particular compilation unit such as `static` globals must have unique names. Because binary linking has these same issues, much of the renaming has already occurred. MIRV prepends the source file name to `static` symbol names to guarantee uniqueness across compilation units. Local symbols are prepended with the procedure name to make them unique across procedures.

The primary difficulty in the IR linker is the naming of types. The C language defines a number of well-known types such as `int` and `char`. These are easy to contend with in the linked environment. More challenging is the use of anonymous structure types as shown in figure 3.47. The type of `global` is unnamed. It only appears once in the C source but because of MIRV's restriction that all type declarations must appear before variable declarations that use the type the compiler must give it a name. Within each compilation unit, MIRV keeps a counter to uniquely name such anonymous structs. For example, the above type might be named `anon.struct_1`.

The MIRV IR linker attempts to compact the number of type symbols generated by

```
struct tag {
  int a;
  struct tag *next;
};
```

Figure 3.48: A Recursive `struct` Type

matching up types with identical names. This scheme ensures that, for example, only one `int` type is ever declared. This is particularly crucial for phases such as alias analysis that rely on type information for correctness. If the compiler can guarantee that unique types only have one symbol to represent them, pointer comparison can be used to identify unique symbols rather than more expensive symbol structure comparison.

To deal with anonymous types, MIRV encodes the type structure itself into the symbol name. The anonymous struct of figure 3.47 may therefor be given a name such as `struct_anonymous_sint32_a_sint32_b`. It is important to include the name of each field as well as the type so that field reference operations will make sense. An anonymous struct containing fields name `c` and `d` is a different type than that of figure 3.47.

With the above scheme the linker will combine anonymous structs declared in separate compilation units. This is allowed under the ANSI C standard which says that pointers to a `struct` type may be converted to pointers of another `struct` type if the alignment requirements of the second type are no more restrictive than the first [21]. Recursive `struct` types such as that of figure 3.48 are not a problem because it is not possible to declare a pointer to anonymous `struct` types except at the point of variable declaration because there is no tag name associated with the `struct`.

Once the linker has performed the necessary renaming and symbol matching, filters may operate on the IR tree just as if they were operating on a single compilation unit. The `--wholeProgram` command-line argument informs the compiler that it has the entire source

of the program available and may perform inter-procedural analysis and transformation. It is up to the individual filter designer to specify how such actions are performed for a particular analysis or transformation pass. All of the interfaces described in chapters chapter 2, 3 and 4 are available.

In this way, whole-program operation of the compiler looks identical to the separate compilation environment. Additional interfaces may become available if the filter designers provide them but the general mechanisms for program analysis and manipulation may be used. This inter-procedural operation does not require that the user learn a new set of programmer interfaces as would be the case if, for example, inter-procedural operation were performed on program binaries as has been done in some studies [6, 45].

3.4 Conclusion

The MIRV dataflow architecture is build around the Attribute Flow pattern, an extension of the Visitor pattern to incorporate the passing of attribute information through a tree structure. Attribute Flow was inspired by the attribute systems provided by parser tools such as YACC, PCCTS and Spirit [22, 46, 47].

3.4.1 Future Directions

There are many improvements that can be made to the existing dataflow architecture. Probably the most glaring deficiency is in the handling of unstructured code. At the time of this writing, the MIRV dataflow framework does not sufficiently handle functions with unstructured control flow. The framework simply detects when unstructured control is present and skips analysis of the function.

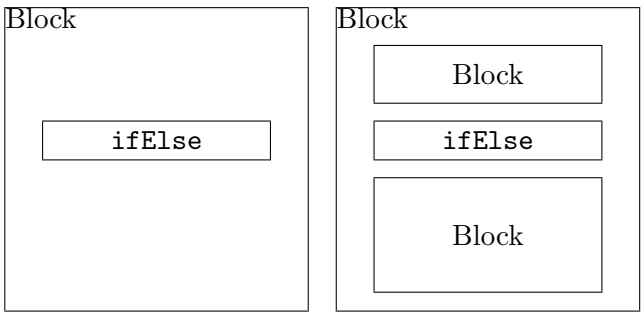
Unstructured code could be handled in a variety of ways within the framework. One

option is to convert the code to a structured form as in the work of Erosa, *et al.* [13]. MIRV currently has a transformation filter which implements the most common cases of unstructured `goto` statements. Another option is to handle the `goto`/label pair by annotating the current attribute context onto each type of statement. The corresponding statement can then pull the dataflow information from its counterpart to correctly compute the necessary confluence. This will require additional code in the existing flow objects.

The current Attribute Flow implementation is quite inefficient in that it always visits every node in the IR on each pass through the program. Most dataflow information changes when sequencing through definition points in the subject program. Likewise, control split and join points also affect the dataflow information propagated to a particular program point. Expressions that simply use data generally do not affect the attribute values flowing through the program. This is the case for all of the current analyses in the MIRV compiler.

We can use this observation and an idea from classical iterative dataflow algorithms to improve the efficiency of attribute flow. Classical iterative dataflow traverses each basic block in the subject program once to compute two sets of information, a *gen* set specifying any new dataflow information computed in the block and a *kill* set specifying which dataflow information should be removed from the set propagated to the block. Once these sets are computed the algorithm need only flow through the program at the basic block level, ignoring individual instructions while iterating to compute the fixed-point solution. Once the solution is obtained the algorithm once again flows through each block to propagate the information to each instruction, killing the appropriate information as data is defined within the block.

The MIRV framework can take the same approach. Expression trees may be visited in a separate pre-pass to compute use information. The Attribute Flow object then need



(a) Before Splitting (b) After Splitting

Figure 3.49: Block Splitting

only flow over statements, ignoring the expression trees below them. This will speed up the iteration process by eliminating the visiting of a significant amount of code.

In effect the classical algorithm is using the same statement/expression observation made above. The classical algorithms recognize that flow through the program need only be concerned about control flow and that individual definition points can be handled in a post-pass. We can use this observation to further improve analysis efficiency. If Attribute Flow flows only to block statements we can employ the same strategy to avoid visiting more code. This model requires that some block statements be "split" as shown in the example of figure 3.49. This is because the high-level nature of the IR does not explicitly encode all of the control join points in a program. If we are only annotating block statements we must make new block "anchor" points for the results of dataflow confluence at these points.

The current framework uses an iterative approach to implement the Attribute Flow pattern. A more natural implementation would use structural dataflow to compute information at each program point [11, 10]. In this model dataflow equations are associated with high-level tree structures. These equations are used to compute the dataflow produced by an entire program subtree given some input dataflow information. The primary complication

of this approach is determining the dataflow equations for each type of analysis and control structure. For structured forms is is fairly straightforward but unstructured code results in complex formulas.

The iterative Attribute Flow implementation presents an opportunity to compute the structural dataflow equations in a systematic way. The problem can be considered another program analysis to which Attribute Flow may be applied. Each action `execute` method would simply contribute its part of the equation to the larger equation being built for its argument's parent node given the sub-equations produced by the argument's child nodes. In this way synthesized attributes are used to build up equations for a statement while inherited attributes account for the sequencing between statements. Confluence operations simply combine two sub-equations in the necessary fashion depending on the analysis and control flow. Because the iterative approach naturally handles unstructured code, the fixed-point solution results in the dataflow equations for the particular region being analyzed.

Structural dataflow equations can be efficiently represented in C++ with expression templates [48]. This representation may allow optimization of the equations themselves as the equation itself is represented as a hierarchy of objects. Just as expression trees may be optimized via constant folding, common subexpression elimination and other classical transformations, so to may these equation objects be optimized.

The MIRV analysis infrastructure presents a new twist on older tried-and-true techniques such as iterative and structural dataflow analysis. These older techniques have been or can be adapted to work on a high-level IR such as MIRV. MIRV takes an evolutionary rather than revolutionary approach. In chapter 4 we will see that the transformation architecture builds upon material presented here, providing a consistent interface to the compiler developer.

CHAPTER 4

MIRV Transformation Model

4.1 Introduction

Chapter 3 described the model of program analysis provided by the MIRV compiler framework. This chapter concentrates on the MIRV program transformation model. We begin with an overview of program transformation and its requirements. Following that we present the MIRV transformation API and comment on some of its unique features. As in the dataflow model description, we present a full example of a program transformation as currently implemented in the MIRV compiler.

4.2 Transformation Overview

A *program transformation* is the changing of a program's syntactic structure. Most often this takes the form of *code motion*, the reordering of code fragments or *code elimination*, the removal of code fragments from the program. Compilers perform code transformations to improve some aspects of program behavior. Common goals include increasing program execution speed and reducing the static code size of the program.

In the vast majority of cases, program transformations must be semantically transparent.

That is, they are not allowed to change the meaning or result of a program. Doing so would present surprising and often incorrect results to the user.

To maintain program correctness, a transformation pass must consider and preserve program semantics. Because programs manipulate various pieces of data, the relationships between those pieces and how they are transformed by each program point must be understood. This is precisely the information dataflow analysis provides. Thus program transformation is most often strictly dependent on dataflow analysis, the specific type of analysis being directly tied to the type of transformation under consideration.

A common example is the restriction that code motion cannot move a use of some piece of data before its definition in the dynamic execution of the program. Reaching definition analysis allows the transformation pass to determine if it is about to violate this requirement. Examination of the U-D chains at the point being moved will indicate the definitions beyond which it may not move.

Transformation passes must also be concerned with how they affect the dataflow information. In the code motion example, any movement necessarily changes the reaching definition information because the location of either the relevant definitions or uses has changed. In the case of code elimination either definitions or uses may disappear entirely. Thus the transformation framework must be capable of recognizing when the dataflow information has be come stale and take action to rectify the situation.

In the next section we present the MIRV transformation architecture and programmer interface. Through the use of design patterns similar to those in the dataflow architecture, MIRV is able to present a view of program transformation that allows the designer to concentrate only on those aspects of the program in which he is interested. Interfaces are provided to signal when dataflow information needs update and in some cases provides the

ability to update information on-the-fly as transformations are being performed, avoiding the extra overhead of executing a dataflow analysis pass from scratch.

4.3 Transformation Architecture

A transformation filter works like an analysis filter except the action objects are not concerned with manipulating dataflow information. Rather, the action objects examine node attributes corresponding to the dataflow information they need to guarantee a safe transformation. If a transformation is safe and the filter believes it to be beneficial, the transformation will be performed.

Given this short description and the previous introductory background we can generate a list of tasks that transformation filters must perform:

1. Traverse the IR
2. Identify potentially beneficial transformations
3. Query dataflow information to guarantee safety
4. Perform the transformation
5. Update or invalidate dataflow information

The MIRV transformation architecture provides generic methods of performing the first, fourth and fifth tasks. The identification potential transformations is entirely dependent on the type of transformation being considered so the developer must specify how this is done. The methods to query dataflow information are provided by the dataflow node attributes themselves and thus cannot be defined within the transformation architecture.

```
void mirvTreeFlow::visit(mirvWhileStatement* node)
{
  mirvFlow::flowState oldState = getFlowState();
  attributeManager.enterNode();
  beforeAction.execute(node);

  setFlowState(mirvFlow::normal);
  node->getConditionStatement()->accept(*this);

  node->getWhileBody()->accept(*this);

  setFlowState(oldState);
  afterAction.execute(node);
  attributeManager.exitNode();
}
```

Figure 4.1: `mirvTreeFlow::visit(mirvWhileStatement *)` Implementation

Often a transformation filter will iterate over this series of actions multiple times for a single filter invocation. This is most frequent when the dataflow information is simply invalidated because the filter must re-invoke the relevant dataflow analysis filters to generate up-to-date information before it can perform any more transformations.

4.3.1 IR Visitation

The transformation architecture is built upon the same Attribute Flow pattern as the dataflow architecture. This consistency simplifies the task of reasoning about how transformation filters do their job. The same techniques of category matching in the filter actions can be used to focus the programmer's effort on the relevant portions of the IR tree. In the vast majority of cases transformation filters do not need attribute propagation and thus set up an attribute manager that simply propagates empty attributes. However occasionally attributes are convenient to use and providing that flexibility comes at no programmer cost thanks to the separation of concerns provided by Attribute Flow.

The transformation filters in general exhibit one large difference from the analysis filters in their behavior: they usually do not iterate over looping structures. Because the trans-

formation filters only manipulate code and in general do not attempt to analyze it, there is no need to come to a fixed-point solution to an analysis problem. Each transformation happens only once so there is no need to go back and re-examine the situation[1].

Given this simplified IR traversal, the MIRV framework provides the `mirvTreeFlow` class. `mirvTreeFlow` is entirely analogous to `mirvForwardFlow` and `mirvBackwardFlow` in that it derives from `mirvFlow` and implements the visitation of statement structures. `mirvTreeFlow` is implicitly a forward traversal because transformations usually do not depend upon the sequence of actions. Any future transformations that do have such dependencies can be implemented through the creation of a new flow class. Figure 4.1 shows the `mirvTreeFlow` visitation of a `while` loop. The flow simply visits the loop condition and body in sequence and does not iterate.

4.3.2 Performing Transformations

In order to effect a code transformation the filter must know what node to manipulate and how to go about doing the needed manipulation. In general transformations take one of two forms: code reorder and code removal. Almost all code transformations can be described in one of these two ways. Further, these forms can themselves be broken down into two subtasks: code replacement and code insertion. Beyond that, we can reason about what these actions mean for MIRV IR constructs.

Recall that the MIRV IR defines two broad categories of code: expressions and statements. Each category has its own unique qualities. Statements can alter program state. Therefore they provide separation between distinct value computations. Statements do not generate any values, they simply sequence the code that does value computation. Therefore

[1] Other than to recompute any needed dataflow information, of course

statements may be removed entirely if the computation they contain is not needed. Expressions by definition cannot alter machine state. They simply compute values. It follows that an expression cannot be removed entirely because it provides a value needed by code further up the IR tree[2].

Code Reorder	
Statements	*Expressions*
Insert copy before new location Replace old copy with null statement	Swap expressions at new and old locations
Code Removal	
Statements	*Expressions*
Replace with null statement	Replace with equivalent simplified expression

Table 4.1: Code Transformation Implementations

Given these constraints we can define what code reorder and removal means for statements and expressions along with how to go about performing it using the replace/insert idiom. Table 4.1 describes the set of actions needed to effect reorder and replacement for statements and expressions. MIRV provides an interface to perform replacement and insertion in a generic fashion. Given these two interfaces most code transformations can be implemented in a straightforward manner according to the rules of table 4.1.

The replace/insert interface is provided through the use of *replacement attributes*. Replacement attributes are node attributes that retain information about how a particular piece of code may be manipulated within the IR tree structure. When a code transformation is being written, the developer usually wants to think about it at the node being manipulated. For example, dead code elimination of an assignment statement should be expressed in the `execute` member of the dead code elimination filter's action object. Therefore, the replacement attribute describes how a node can manipulate itself within the IR tree. In order to do this the node needs to know two things: how to get at its parent and

[2]Unless the entire statement containing the expression is removed, of course.

```
class mirvReplacementActionAttribute : public mirvNodeAttribute {
public:
  mirvReplacementActionAttribute();
  mirvReplacementActionAttribute(const mirvReplacementActionAttribute&);
  mirvReplacementActionAttribute(mirvReplacementAction* f);
  void operator=(const mirvReplacementActionAttribute& r);
  virtual ~mirvReplacementActionAttribute();

  virtual void replace(mirvCode* mc) const;
  virtual void insertAfter(mirvCode* mc) const;
  virtual void insertBefore(mirvCode* mc) const;
  virtual mirvCode* getParent() const;

  // Static version of above methods.  These extract the replacement
  // attribute automatically. They are deprecated by mirvCode interfaces.
  static void replaceWith(mirvCode* replacee, mirvCode* replacement);
  static void insertAfter(mirvCode* c, mirvCode* after);
  static void insertBefore(mirvCode* c, mirvCode* before);

  // Backward IR tree traversal
  static mirvCode* getParentOf(const mirvCode* c);

  mirvReplacementAction* getReplacementAction() const;
  // Check if the Action is valid (ie, not null)
  bool isValid() const;

  mirvReplacementActionAttribute *clone(void) const;

private:
  mirvReplacementAction* Action;
};
```

Figure 4.2: Replacement Attribute Model

where within the parent structure the reference to the node exists. Replacement attributes go just a little further than this to provide more convenience to the developer.

The `mirvReplacementAttribute`, shown in figure 4.2 provides the interface for code manipulation. To do its job the attribute holds a reference to a *replacement action* object, modeled in figure 4.3. The replacement action does the code manipulation grunt-work. It inherits from `mirvAction` only as a historical oddity. The current attribute-based replacement implementation removes this requirement. The `execute` method is a synonym for `replace` and is similarly deprecated.

```
class mirvReplacementAction : public mirvAction {
public:
  mirvReplacementAction(mirvCode *parent, mirvCode *me);
  virtual ~mirvReplacementAction(void) { parent = 0; me = 0; };

  mirvCode* getParent(void) const;
  void setParent(mirvCode *p);

  virtual void replace(mirvCode* mc);
  virtual void insertAfter(mirvCode* mc) = 0;
  virtual void insertBefore(mirvCode* mc) = 0;
  virtual mirvReplacementAction* clone() const = 0;

  // Deprecated
  virtual void execute(mirvCode *mc) = 0;

protected:
  mirvCode* parent;
      mirvCode *me;
};
```

Figure 4.3: Replacement Action Model

In order for the replacement action to perform code manipulation, it needs to have three pieces of information: a reference to the code being manipulated, a reference to the parent of that code subtree and a reference to the proper parent member function to effect the desired code transformation. For example, to support `ifElse` condition expression replacement the replacement action must keep a reference to the `setCondition` method of the `mirvIfStatement` class. Fortunately, C++ provides the member function pointer type that makes this possible.

It is easy enough for the replacement action of figure 4.3 to hold two `mirvCode` references to parent and child but it is not immediately clear how to represent the needed member function references. This is because the type interface to the functions changes based on the type of node being considered. For example, `mirvIfStatement::setCondition` expects to received a `mirvExpression` pointer. On the other hand, `mirvWhileStatement::setBody` expects to received a `mirvStatement` pointer. Clearly we cannot represent all possible

```
template <class TParent, class TChild>
class mirvDirectReplacementAction : public mirvReplacementAction
{
public:
  typedef void (TParent::*replFuncType)(TChild *);

  mirvDirectReplacementAction(TParent *parent, TChild *me, replFuncType replacementFun);
  virtual ~mirvDirectReplacementAction(void) {};

  virtual void insertAfter(mirvCode* mc);
  virtual void insertBefore(mirvCode* mc);
  virtual mirvReplacementAction* clone() const;

  virtual void execute(mirvCode *mc);

protected:
  replFuncType replacementFunction;
};
```

Figure 4.4: Direct Replacement Action Model

```
template <class TParent, class TChild>
void
mirvDirectReplacementAction<TParent, TChild>::execute(mirvCode *mc)
{
  (((TParent *) parent)->*replacementFunction)((TChild *) mc);
}
```

Figure 4.5: Direct Replacement `execute`

interfaces in the `fig:replacementAction` without great complexity.

To solve this problem, the MIRV framework provides several additional classes that derive from `mirvReplacementAction`. The `mirvDirectReplacementAction` class, the simplest model of the replacement action, is shown in figure 4.4. The direct replacement action holds a pointer to a function that replaces one `mirvCode` with another. This function is "direct" in that the function itself knows which piece of code it is replacing (`setCondition` or `setLeftOperand`, for example). The class has two template parameters: the type of the parent node and the type of the child node to which this attribute will be attached. This allows type-safe code replacement[3].

[3]The current implementation requires casting because the base `mirvReplacementAction` class holds `mirvCode` pointers. This can be eliminated by storing the parent/child pointers in the subclasses.

```
template <class TParent, class TChild>
void mirvDirectReplacementAction<TParent, TChild>::insertBefore(mirvCode* mc)
{
  mirvStatement *newStmt =
    dynamic_cast<mirvStatement *>(mc);
  assert(newStmt != 0);
  me->getParentStatement()->insertBefore(newStmt);
}
```

Figure 4.6: Direct Replacement `insertBefore`

```
template <class TParent, class TChild>
void
mirvDirectReplacementAction<TParent, TChild>::insertAfter(mirvCode* mc)
{
  mirvStatement *newStmt =
    dynamic_cast<mirvStatement *>(mc);
  assert(newStmt != 0);
  me->getParentStatement()->insertAfter(newStmt);
}
```

Figure 4.7: Direct Replacement `insertAfter`

The implementation of the direct replacement attribute is presented in figures 4.5, 4.6 and 4.7. Each method does a sanity check on the code to be added to make sure it adheres to the MIRV IR requirements. For example, only statements can be inserted because inserting an expression would live the result value dangling in the IR tree. To avoid the overhead of `dynamic_cast` the attributes could require that `insertBefore`/`insertAfter` receive `mirvStatement` pointers. The current interface is a historical artifact.

The direct replacement action is assumed to operate on parent nodes that are expressions. Therefore, inserting code before or after them is nonsensical. To resolve this problem the framework assumes the programmer really wants to insert the statement before or after the most immediate parent statement of the expression. In practice these methods are never called by transformation filters because the filters generally will never want to insert something while operating within an expression. This is entirely due to the natural semantics of code transformation.

```
template <class TParent, class TChild>
class mirvDirectStatementReplacementAction
  : public mirvDirectReplacementAction<TParent, TChild> {
public:
  typedef void (TParent::*replFuncType)(TChild *);

  mirvDirectStatementReplacementAction(TParent *parent, TChild *me,
                                       replFuncType replacementFun);
  virtual ~mirvDirectStatementReplacementAction(void) {};

  virtual void insertAfter(mirvCode* mc);
  virtual void insertBefore(mirvCode* mc);
  virtual mirvReplacementAction* clone() const;
};
```

Figure 4.8: Direct Statement Replacement Action Model

```
template<class TParent, class TChild>
void mirvDirectStatementReplacementAction<TParent, TChild>::
insertBefore(mirvCode* mc)
{
   // Replace this with a block, insert this, then insert the new statement
   mirvBlockStatement *newBlock = new mirvBlockStatement;
   assert(me != 0);
   mirvStatement *temp = dynamic_cast<mirvStatement *>(me)->clone();
   replace(newBlock);
   newBlock->statementsPushFront(temp);
   newBlock->statementsPushFront((TChild *)mc);
}
```

Figure 4.9: Direct Statement Replacement `insertBefore`

```
template<class TParent, class TChild>
void mirvDirectStatementReplacementAction<TParent, TChild>::
insertAfter(mirvCode* mc)
{
   // Replace this with a block, insert this, then insert the new statement
   mirvBlockStatement *newBlock = new mirvBlockStatement;
   assert(me != 0);
   mirvStatement *temp = dynamic_cast<mirvStatement *>(me)->clone();
   newBlock->statementsPushFront(temp);
   newBlock->statementsPushBack((TChild *)mc);
   replace(newBlock);
}
```

Figure 4.10: Direct Statement Replacement `insertAfter`

```
template <class TParent, class TIndex, class TChild>
class mirvIndirectReplacementAction : public mirvReplacementAction {
public:
  typedef void (TParent::*replFuncType)(TIndex, TChild*);

  mirvIndirectReplacementAction(TParent *parent, TChild *me,
                                replFuncType replacementFun, TIndex index);
  virtual ~mirvIndirectReplacementAction(void) {};

  virtual void insertAfter(mirvCode* mc);
  virtual void insertBefore(mirvCode* mc);
  virtual mirvReplacementAction* clone() const;

  virtual void execute(mirvCode *mc);

protected:
  TIndex index;
  replFuncType replacementFunction;
};
```

Figure 4.11: Indirect Replacement Action Model

Figure 4.8 implements replacement on statements for direct-access replacement member functions. Figures 4.9 and 4.10 show the implementations of code insertion. Replacement operates as in `mirvDirectReplacementAction`. Both insert routines use a similar strategy: the existing statement is replaced with an empty block statement. The previous statement is inserted into the block and the new statement is inserted before or after the old statement as needed. This strategy is used because there is no guarantee that the old statement is itself a block statement. Making a new empty block simplifies the insertion routine.

The indirect replacement action of figure 4.11 is only slightly more complex than the direct variant. Indirect replacement is used in situation where the code manipulation routines operate upon lists of code, as in a block statement. In those cases the replacement attribute needs to know the position of the code being manipulated within the list. The new `Index` template parameter is the type of this position information. It is generally an iterator into a code list, for example an iterator into the list of statements contained within a `mirvBlockStatement` object.

```
template<class TParent, class TIndex, class TChild>
void
mirvIndirectReplacementAction<TParent, TIndex, TChild>::execute(mirvCode *mc)
{
  assert(*index == me);
  (((TParent *) parent)->*replacementFunction)(index, (TChild *) mc);
}
```

Figure 4.12: Indirect Replacement `execute` Implementation

The `execute` method of the indirect replacement action is shown in figure 4.12. Because the indirect replacement action does not hold a reference to an insert method it cannot perform insertion transformations. The `insertBefore` and `insertAfter` implementations simply abort compiler execution. This is an unfortunate consequence of the IR design. Certain statements such as case statements within a switch statement cannot be of `mirvBlockStatement` type. This is because the statements must be of a type (such as `mirvCaseStatement`) that carries extra information necessary for proper code generation (switch labels, in this example). The insert routines could not simply create an empty block statement in the the direct replacement case. Direct replacement attributes are never attached to nodes that cannot be of `mirvBlockStatement` type.

The `mirvIndirectStatementReplacementAction` class of figure 4.13 is provided to effect code insertion at the statement level. The `insertBefore` method is given in figure 4.14. The attribute holds a reference to the function to do replacement. Classes that contain lists of statements have methods to insert into the list before an iterator of the list, similar to the standard C++ library's `std::list` interface. To provide the `insertAfter` manipulation the replacement action simply increments the index/iterator before calling the insertion routine.

Given a replacement attribute the programmer can call `replace` to substitute the given code for the code to which the replacement attribute was attached. Similarly, the program-

```
template <class TParent, class TIndex, class TChild>
class mirvIndirectStatementReplacementAction
  : public mirvIndirectReplacementAction<TParent, TIndex, TChild> {
public:
  typedef TIndex (TParent::*insertFuncType)(TIndex, TChild*);

  mirvIndirectStatementReplacementAction(TParent *parent, TChild *me,
                                         replFuncType replacementFun,
                                         insertFuncType insertionFun,
                                         TIndex index);
  virtual ~mirvIndirectStatementReplacementAction(void) {};

  virtual void insertAfter(mirvCode* mc);
  virtual void insertBefore(mirvCode* mc);
  virtual mirvReplacementAction* clone() const;

protected:
  insertFuncType insertionFunction;
};
```

Figure 4.13: Indirect Statement Replacement Action Model

```
template<class TParent, class TIndex, class TChild>
void
mirvIndirectStatementReplacementAction<TParent, TIndex, TChild>::insertBefore(mirvCode*
mc)
{
  (((TParent *) parent)->*insertionFunction)(index, (TChild *) mc);
}
```

Figure 4.14: Indirect Statement Replacement `insertBefore` Implementation

```
void mirvReplacementVisitor::visit(mirvWhileStatement* node)
{
  visitSingle(node->getConditionStatement(),
              new mirvDirectReplacementAction<mirvWhileStatement,
                  mirvExpressionStatement>(node, node->getConditionStatement(),
                                           &mirvWhileStatement::
                                            setConditionStatement));

  visitSingle(node->getWhileBody(),
              new mirvDirectStatementReplacementAction<mirvWhileStatement,
                  mirvStatement>(node, node->getWhileBody(),
                                 &mirvWhileStatement::setWhileBody));
}
```

Figure 4.15: Setting `while` Statement Replacement Attributes

```
void mirvReplacementVisitor::visit(mirvBlockStatement* node)
{
  for(mirvBlockStatement::statementIterator i = node->statementsBegin();
      i != node->statementsEnd();
      i++) {
    visitSingle(*i,
                new mirvIndirectStatementReplacementAction<mirvBlockStatement,
                    mirvBlockStatement::statementIterator,
                    mirvStatement>(node, *i,
                                   &mirvBlockStatement::setStatement,
                                   &mirvBlockStatement::statementsInsertBefore,
                                   i));
  }
}
```

Figure 4.16: Setting Block Statement Replacement Attributes

mer can insert code before or after the annotated code via `insertBefore` and `insertAfter`, respectively. Due to the implementation of the replacement action object invocation of insert methods on an expression node will insert the given code before or after the lowest-level statement containing the expression.

Replacement attributes are themselves set by a special filter. This filter is nominally an analysis filter because it must determine which method interfaces to include in the attributes. Figures 4.15 and 4.16 show how the direct and indirect replacement attributes are attached to the IR tree. Functions called by these methods are given in figure 4.17. In each case the replacement attribute is constructed with the proper code and member

```
void  mirvReplacementVisitor::
visitSingle(mirvCode* single, mirvReplacementAction* f)
{
  setReplacementAction(single, f);
  single->accept(*this);
}

void mirvReplacementVisitor::
setReplacementAction(mirvCode* node, mirvReplacementAction* f)
{
  mirvNodeAttribute*& ra = node->nodeAttribute(Replacement);
  if (ra != 0) { delete ra; }
  ra = new mirvReplacementActionAttribute(f);
}
```

Figure 4.17: Replacement Filter Helpers

function pointers to effect the replacement and insertion operations.

The replacement attribute concept decouples the transformation interface from the IR tree proper. Tree nodes need not concern themselves with how they relate to their parents. The replacement attribute contains the necessary information. Transformation filters can concentrate their efforts on the nodes being manipulated without needing to be aware of the IR context in which those nodes appear.

As an interesting side note, there are no back-pointers in the MIRV IR API. That is, node classes do not contain pointers to their parent nodes. The replacement attributes have all of the necessary information to walk backward through the IR tree. Figure 4.18 shows the implementation of `mirvCode::getParentStatement`. It simply queries the replacement attribute for the parent node and checks whether it is a statement. If not it recursively explores the parent node until a statement is found or the top of the IR tree is encountered.

4.3.3 Dataflow Patching

As we have seen in this example, the dead code elimination filter is able to update existing dataflow information as it performs its task. There is direct support for updating

```
mirvStatement *mirvCode::getParentStatement(void) const
{
  if (hasNodeAttribute(Replacement)) {
    mirvCode *parent = mirvReplacementActionAttribute::getParentOf(this);

    mirvStatement *stmt = dynamic_cast<mirvStatement *>(parent);
    if (stmt != 0) {
      return(stmt);
    }
    else {
      return(parent->getParentStatement());
    }
  }
  else {
    return(0);
  }
}

mirvCode*
mirvReplacementActionAttribute::getParentOf(const mirvCode* c)
{
  const mirvReplacementActionAttribute* rfa =
    dynamic_cast<const mirvReplacementActionAttribute*>(
      c->nodeAttribute(Replacement));

  assert (rfa != NULL);

  return rfa->getParent();
}
```

Figure 4.18: Backward IR Tree Traversal

reaching definition information in the MIRV framework. This is a prototype implementation and could be extended to other types of dataflow as well. This will require a smarter system to allow easy incorporation of new analysis information into the up-to-date bit-vectors. Moreover the `removeDataflow` and `clone` interfaces must be flexible enough to allow the programmer to specify different types of dataflow to be manipulated. Currently these interfaces rely on the reaching definition attribute interface. A better design would place the update code into the attribute itself, freeing the IR code classes from needing any prior knowledge of dataflow information structure.

This is an active area of development within the MIRV framework. Day-to-day use has demonstrated the utility of the dataflow patching approach. Compile times can be reduced significantly if repeated dataflow analyses over the entire function can be avoided.

4.4 An Example: Dead Code Elimination

As an example of the transformation API we present the MIRV implementation of Dead Code Elimination. Dead code elimination removes code that is no longer needed either by identifying data definitions that are not used anywhere or by determining that a piece of code can never execute. As in the analysis filter case, we can identify several tasks that the transformation filter designer must accomplish:

1. Determine what dataflow information is necessary to maintain correctness
2. Determine which types of nodes are of interest and implement action methods for them
3. Package the objects into the Attribute Flow framework and invoke the transformation
4. Invalidate or update dataflow information so that later passes can detect the change

```
class mirvDeadCodeVisitor : public mirvActionVisitor {
public:
  mirvDeadCodeVisitor(int i);
  virtual ~mirvDeadCodeVisitor() {};

  // ... Statistics routines

  bool wasChanged(); // Was the mirv tree been changed by the visitor?

  void visit(mirvIfElseStatement*);
  void visit(mirvAssignStatement*);
  // ... Other dead code nodes

private:
  // ...
};
```

Figure 4.19: Dead Code Action Class

To maintain program correctness, we must be sure not to eliminate any code that can affect machine state that is user-visible. This means that we cannot safely remove definitions that may be used elsewhere in the code. Therefore we rely on reaching definition information computed in an earlier analysis pass.

Determining nodes of interest is straightforward. Code can be eliminated under two conditions: if definitions in the code do not reach anywhere or if the code cannot execute for some reason. Code may not execute for a variety of reasons. It may be guarded by a conditional statement that is always false, there may be an unconditional branch before the code so that it is never reached or the function in which the code lives may never be called. To simplify the presentation we will only concern ourselves with dead assignment statements and `ifElse` statements for which conditions can be statically determined. The actual filter currently available in the MIRV compiler handles many more cases of dead code.

Now that we have determined which types of IR constructs we may eliminate or simplify, we can implement action routines for them. As with analysis filters, we define a visitor to

```
void mirvDeadCodeVisitor::visit(mirvAssignStatement* e)
{
  // Get the Assignment's Def Attribute and Reaching Def attribute
  if (!e->hasNodeAttribute(Def)) {
    return;
  }

  mirvDefAttribute defAttr =
    e->template getInternalNodeAttribute<mirvDefAttribute>(Def);

  // Do not eliminate assignments to globals or arrays
  for(mirvDefAttribute::dataIterator v = defAttr.dataBegin();
      v != defAttr.dataEnd();
      v++) {
    if ((*v)->getDataType() != mirvData::local) {
      return;
    }
    if ((*v)->hasArrayType()) {
      return;
    }
  }

  // Check if it's never used.
  // Also, check if the only uses are in the assignment itself.
  bool usesOnlyInAssign = true;
  for(mirvDefAttribute::useIterator u = defAttr.usesBegin();
      u != defAttr.usesEnd();
      ++u) {
    if ((*u)->getCode()->getParentStatement() != e) {
      usesOnlyInAssign = false;
      break;
    }
  }

  // ... more
```

Figure 4.20: Dead Code Assignment Action, Part 1

be used with a flow object. This is shown in figure 4.19. As in the live variable analysis design, we make use of the `mirvActionVisitor`/`mirvVisitorAction` glue as a backward-compatibility bridge.

The filter action for assignment statements is given in figures 4.20 and 4.21. The presentation is a simplified version of the actual code in the MIRV compiler. The additional code handles special cases such as speculative register promotion [49].

```
  // ... more

  if (usesOnlyInAssign) {
    // Check if RHS has side effect (i.e., function call)
    mirvFunctionExpression *call =
      dynamic_cast<mirvFunctionExpression *>(e->getRightOperand());
    if (call != 0) {
      // Eliminate the return value assignment
      if (preserveDataflow) {
        mirvStatement *newStmt = new
mirvExpressionStatement(e->getRightOperand()->clone(/*copyDataflow = */true));
        e->removeDataflow();
        e->replaceWith(newStmt, /*setChanged =*/false);
      }
      else {
        mirvStatement *newStmt = new
mirvExpressionStatement(e->getRightOperand()->clone());
        e->replaceWith(newStmt);
      }
    }
    else {
      // Eliminate the statement entirely
      if (preserveDataflow) {
        e->removeDataflow();
        e->replaceWith(new mirvNullStatement, /*setChanged =*/false);
      }
      else {
        e->replaceWith(new mirvNullStatement);
      }
    }
  }
}
```

Figure 4.21: Dead Code Assignment Action, Part 2

The first thing to do is check that the proper dataflow information is available. If not we simply avoid performing any transformation. If the information is available, we check to make sure we're not defining some data for which we cannot see all of its uses. An example is the definition of a global variable. Since we are not performing inter-procedural transformations we cannot know if the global might be used elsewhere. Array elements also fall into this category because MIRV does not yet treat each element in an array separately.

The next loop checks to see if the definition is actually used. Even if the definition is only used within the statement itself (for example, a loop variable increment) we may still eliminate it if there are no uses outside the statement[4].

If the assignment is determined not to reach elsewhere there is one more check to be performed. The right-hand-side of the statement may be a function call which can produce side-effects. If so we cannot eliminate it. A simple `dynamic_cast` suffices because the MIRV IR guarantees that a function call may appear only as a separate statement or the immediate right-hand-side of an assignment. The Reaching Definition filter annotates (potential) function call definitions on the call node itself rather than on the statement containing the call. If the definitions were added to the attribute of the latter the semantics of the call would be visible at the assignment statement and we could eliminate this extra side-effect check. If the check finds a function call we may replace the assignment statement with the function call, eliminating the return value assignment.

Once we determine that code may be removed, we invoke some routines from `mirvCode` to do the work. The Dead Code Elimination filter is smart enough to update reaching definition dataflow information if told to do so. The flag `preserveDataflow` is maintained by the filter and passed to the action objects so they may know how to proceed. If dataflow

[4]The induction variable elimination filter can help eliminate code with multi-statement loop-carried dependencies.

```
void mirvCode::replaceWith(mirvCode *e, bool setChanged)
{
  assert(hasNodeAttribute(Replacement));

  mirvCode *parent = getParent();

  const mirvReplacementActionAttribute &replace =
    this->template getInternalNodeAttribute<mirvReplacementActionAttribute>(Replacement);

  if (setChanged) {
    mirvFunctionSymbol *function = getParentFunction();
    if (function) function->changed();
  }
  else {
    mirvFunctionSymbol *function = getParentFunction();
    if (function) function->mostlyChanged();
  }

  // Kills this!
  replace.replace(e);
  // We should re-run replacement annotation.
  mirvReplacementVisitor v;
  parent->accept(v);
}
```

Figure 4.22: `mirvCode::replaceWith` Implementation

information is not to be preserved, eliminating code is simply a matter of replacing the assignment with a null statement (or function call if one exists). The `mirvCode::replaceWith` function invalidates reaching definition dataflow information by default. It's implementation is shown in figure 4.22. The `mirvFunctionSymbol` routines `changed` and `mostlyChanged` invalidate node attribute information throughout the function. The function symbol keeps a bit-vector of representing "important" node attributes. The MIRV framework defines which attributes are "important" by whether they present analysis information used to maintain transformation safety. Thus new analyses must be given bits in the bit-vector to represent their up-to-date status. The `changed` routine invalidates all information. The `mostlyChanged` routines invalidates everything but reaching definition information. If the dead code elimination filter is told to preserve dataflow it calls this routine. In addition it

invokes the virtual `mirvCode::removeDataflow` routine. This routine recursively invokes the `mirvCode::removeDefs` and `mirvCode::removeUses` routines to remove the reaching definition information from other D-U and U-D chains in the program.

A different kind of dead code elimination is performed by the code in figure 4.23. This code checks whether the branch condition is a compile-time constant. If so one branch of the statement may be eliminated entirely and the other may be left unguarded. We have removed some special cases from the code to clarify the presentation. The code removed checks for empty then- or else-clauses and replaces the `ifElse` node with a simple `if` node.

A special visitor object is used to evaluate whether the `ifElse` condition is a compile-time constant and if so, whether it is true or false. The visitor maintains an evaluation stack and acts as a post-action in an Attribute Flow invocation[5]. Value computation proceeds as in a Reverse Polish Notation calculator. In fact the visit/compute action is not unlike what a bottom-up expression parser would do. At this point we assume the existence of this visitor and do not discuss it further.

If the condition evaluates true or false we replace the statement with the appropriate single-arm `if` version. If we are preserving dataflow information we must take two steps: we must remove all of the dataflow information corresponding to the arm we are eliminating and when cloning the arm we are going to maintain we must update the existing dataflow to point to this new copy of the code. The various implementations of the virtual `mirvCode::clone` routine include all of the code necessary to do this.

Once the action objects have been designed we must package them up into an Attribute Flow and invoke the transformation. As with the analysis filters, transformation filters are composed by a plugin class, shown in figure 4.24. The `activate` routine is shown in figure

[5]This visitor was written before Attribute Flow was developed and performs its own post-order IR traversal.

```
void
mirvDeadCodeVisitor::visit(mirvIfElseStatement* s)
{

  // Check if the condition is a constant
  mirvDCEvalCondVisitor evalCondVisitor(getFlow()->getCurrentModule());
  evalCondVisitor.evaluate(s->getCondition());

  // Check if condition is always true
  if (evalCondVisitor.isTrue()) {
    // Condition true: Replace ifElse statement with if body

    if (preserveDataflow) {
      mirvStatement *newBody = s->getIfBody()->clone(/*copyDataflow = */true);
      // Remove stale defs and uses.
      s->removeDataflow();
      s->replaceWith(newBody, /*setChanged = */false);
    }
    else {
      s->replaceWith(s->getIfBody()->clone());
    }
  }

  // Check if condition is always false or the block is empty
  else if (evalCondVisitor.isFalse()) {
    // Condition false: Replace ifElse statement with else body

    if (preserveDataflow) {
      mirvStatement *newBody = s->getElseBody()->clone(/*copyDataflow = */true);
      // Remove stale defs and uses.
      s->removeDataflow();
      s->replaceWith(newBody, /*setChanged = */false);
    }
    else {
      s->replaceWith(s->getElseBody()->clone());
    }
  }
}
```

Figure 4.23: Dead Code `ifElse` Action

```
class mirvDeadCodePlugin : public mirvPlugin {
private:
// ...

public:
  mirvDeadCodePlugin() :
    mirvPlugin(PLUGIN_NAME), debugLevel(0), maxIterations(INT_MAX) {};
  virtual ~mirvDeadCodePlugin(void) {};

  filterRunTime  getRunTime(void)  { return duringOpt; }
  filterRunLevel getRunLevel(void) { return functionLevel; }

  void activate(mirvFunctionSymbol *function);
  void registerOptions(optionDatabase *odb);
};
```

Figure 4.24: Dead Code Plugin

```
void mirvDeadCodePlugin::activate(mirvFunctionSymbol *function)
{
  bool changed = false;

  mirvNullDataflowAttributeManager attribMgr;

  mirvDeadCodeVisitor deadCodeVisitor(debugLevel);
  mirvVisitorAction deadCodeAction(deadCodeVisitor);
  mirvNullAction nullAction;

  mirvReplacementPlugin replacementPlugin;
  mirvDefUsePlugin defUsePlugin;
  mirvTreeFlow flow(nullAction, deadCodeAction, attribMgr);

  do {
    filterStat(functionLevel, completeRun, function);
    replacementPlugin.activate(function);
    defUsePlugin.activate(function);

    deadCodeVisitor.reset();
    function->accept(flow);
    changed = deadCodeVisitor.wasChanged();
    // Print the statistics
    deadCodeVisitor.printStatistics();
  } while(changed);
}
```

Figure 4.25: Dead Code `activate` Routine

4.25. The routine builds a flow object using the dead code action as the pre-action and a null action as the post-action. The dead code action could just as easily have been placed in the post-action slot as the order of code transformations does not matter.

The inner loop of the plugin repeated applies program analyses followed by the dead code elimination pass as long as changes are made to the IR. Dead code elimination depends on replacement attributes and reaching definition attributes being available in the IR tree. Therefore it invokes those plugins before starting the elimination process.

Our final task listed above is to update or invalidate dataflow information. This was done by the action object so we have already completed this task. Our dead code elimination filter is now ready for use.

4.5 Dataflow Patching

As we have seen in our example, the dead code elimination filter is able to update existing dataflow information as it performs its task. There is direct support for updating reaching definition information in the MIRV framework. This is a prototype implementation and could be extended to other types of dataflow as well. This will require a smarter system to allow easy incorporation of new analysis information into the up-to-date bit-vectors. Moreover the `removeDataflow` and `clone` interfaces must be flexible enough to allow the programmer to specify different types of dataflow to be manipulated. Currently these interfaces rely on the reaching definition attribute interface. A better design would place the update code into the attribute itself, freeing the IR code classes from need any prior knowledge of dataflow information structure.

This is an active area of development within the MIRV framework. Day-to-day use has demonstrated the utility of the dataflow patching approach. Compile times can be reduced

significantly if repeated dataflow analyses over the entire function can be avoided.

4.6 Conclusion

The MIRV transformation architecture builds upon the analysis architecture presented in chapter 3. Familiarity with Attribute Flow allows the programmer to concentrate efforts on the IR nodes affected by a particular transformation. Our dead code elimination example demonstrated this separation. Other than various utility classes such as logical expression evaluation, the meat of the filter is contained within just two `visit` routines. As demonstrated in this chapter, some transformation filters can be quite easily grasped in the MIRV framework.

CHAPTER 5

Automatic Debugging Tools for Experimental Compiler Developers

5.1 Introduction

Compiler tool-chains are notoriously difficult to design, produce and debug. Their very nature necessitates a set of complex interactions among several program passes, with each pass contributing information and/or manipulating the program input in some way. This complexity had been to the detriment of the computer architecture research community. Because such tools are difficult to produce, very few research compiler tool-sets are available. Only recently has a coordinated effort begun to make such tools widely available through the National Compiler Infrastructure initiative [50, 29].

Much of the research in computer architecture has been performed without considering the role of the compiler. There have been some important exceptions where architecture-compiler synergy has been explored [34]. Such research requires heavy modification of the compiler software, which dramatically increases the possibility of errors.

It is our experience that computer architects in the academic research community are generally wary of heavy software modification. They are used to working inside of relatively

simple-to-understand tools such as machine simulators that closely model the domain with which they are most familiar. However, with the present shift toward more integration between hardware architecture and software techniques, knowledge of the software/hardware interface provided by the compiler is becoming essential. Many architects express the desire to conduct research using optimizing compilers but find the existing tools difficult to master. In fact, the experimenter is in the same position as a compiler developer, except that, perhaps, it is even more important that she or he debug any errors quickly.

As part of an ongoing effort to develop a viable compiler tool-set for computer architects, we have produced a number of tools that can greatly simplify the compiler developer's task. Verification and debugging of compiler components requires a large time investment from the researcher. Therefore, we have concentrated our ancillary development on tools to make this process simpler and more streamlined.

The goals of this chapter are fourfold: to present a set of tools useful for developers working on optimizing compilers, to explain the motivation and development process for these tools, to convey our experiences using these tools in a research environment and to motivate computer architects and research compiler developers to reexamine and improve the compiler tools currently available to them. We believe these tools to be applicable to compiler development in general, not just to our particular tool-chain.

The rest of the chapter is organized as follows. Section 5.2 presents our regression test suite, touching on its development and portability. In section 5.3 we present our tools for rapid characterization of compiler bugs. We have found these tools to be particularly valuable in our research environment. We have benchmarked various compiler tool-sets currently available to computer architects in order to gauge the confidence researchers can place in these tools. We make no claims as to the relative quality of applicability

of these tools but rather present our findings to spur discussion and tool improvement. Our methodology for these tests is presented in section 5.4 and we evaluate the results in section 5.5. Section 5.6 enumerates some previous work in software testing and debugging. We conclude in section 5.7.

5.2 Regression Testing

A time-honored technique for program verification is the use of regression testing. A regression testing system is composed of a number of small test-cases that expose bugs previously found in a system. In the course of running the test, the test harness is provided some method of checking whether the current version of the system being tested satisfies the test requirements. A failure indicates a regression in the system.

5.2.1 Regression Suite

A compiler regression test is typically a small program or program fragment that is fed into the compiler tool-chain. For our purposes, the tests cover correctness. It is also possible to create regression suites to test performance of the compiled code, compilation time and many other metrics. Because our primary concern has been the ease of use of our system by computer architecture researchers and because correctness is a primary factor in ease of use, our focus has been on that domain.

The test harness performs several checks to verify correctness. The primary check occurs against the program output. All output from the program is logged to a file. This file is compared against a previously-generated program output that is assumed to be correct[1]. In addition, the program return code is saved and checked against the return code generated

[1] "Assumed" because it is possible that the software generating this output is itself flawed.

when the reference output was produced. A difference in either one of these values indicates a correctness failure.

Our reference output is generated by a compiler that is assumed to be correct. Because we use SimpleScalar/PISA in most of our research work we have chosen the PISA port of the GNU C compiler version 2.7.2.3 produced at the University of California, Davis as our benchmark [5].

Our regression suite includes 624 test-cases. Most of these were produced by hand as we discovered bugs in the tool-set. For the most part each test covers one primary bug, though most of them can expose multiple failures. The harness runs each of these tests at three optimization levels. The first optimization level (O0) includes only register allocation. The O2 level includes most of the classical compiler transformations such as common subexpression elimination, propagation and various loop optimizations. The highest level of regression (link-O3) includes whole-program source-level linking, analysis and function in-lining.

Two scripts control the testing process. The first, `runtests`, is given a well-known regression suite name to run, for example "regress" for the full suite or "fail" for tests known to fail. It in turn invokes `runtest` which is responsible for compiling the program, running it and checking its output.

The harness is able to run regressions for all of the back-end targets, though reference output must be generated for each individually as it may be dependent on the machine organization (sizes of C types, for example). In particular, it is able to run tests in both "bare metal" and simulated environments. The former is used to test our IA32 back-end and the latter is used for all other architectures.

Most of the test-cases are grouped under a "general" category and are run every time

regression testing is performed. A few test-cases are held for special purposes such as testing specific compiler filters or other situations.

5.2.2 Portability

Our regression testing system has proven to be remarkably portable and applicable to other tool-set environments though it was not originally designed with this purpose in mind. We have ported the system to be used with other research compilers such as lcc and MachSUIF [51, 29]. This has provided us an interesting opportunity to benchmark other compilers available to the computer architecture research community. The results of this benchmarking appear in section 5.5.

The system has also proven useful to validate non-compiler software. In particular, we have used the test-suite to verify machine simulator software. As part of our research into compiler/architecture synergy in modern microprocessor systems, we ported the M5 simulator to the PISA architecture [3]. PISA is a convenient instruction set for compiler developers as it has a large and relatively sparse opcode space, making the addition of compiler-visible ISA changes relatively simple. Once the simulator was ported to PISA we used the regression suite to verify its correctness. However, instead of running a regression test using the MIRV/SimpleScalar system, we ran each test using a gcc/M5 system benchmarked against a gcc/SimpleScalar system. The test-suite exposed many bugs in our port and allowed rapid stabilization of the software, to the point where we were able to conduct useful research within about one week from the completion of the pre-verification porting task.

We have continued to use this setup to validate simulator and PISA ISA changes. For example, we have used the test harness to verify the correctness of a unique cooperative

register allocation strategy initially implemented in SimpleScalar that we have ported to M5 [49]. The regression suite has proven invaluable in quickly testing and debugging new compiler and architectural optimizations.

5.2.3 Larger Benchmarks

The regression suite is useful for quickly verifying compiler correctness, but the small and limited nature of each test-case necessarily limits the scope of testing that can be performed. In particular, it is inconvenient to verify complex whole-program analyses and transformations using such small test-cases.

To alleviate this problem, our test-suite includes scripts to run all of the SPECint95 and SPECint2000[2] benchmarks to verify compiler correctness when working with large programs. The harness is able to run these programs at three different optimization levels. A pool of execution machines greatly reduces the time to finish this testing, though it is of course not a requirement.

5.2.4 Stress Testing

The above sections describe our most common regression tests. These are the tests that are run regularly, at least each time the source repository is about to change. We also include some tools for stress testing to shake out bugs that the regression tests do not uncover. Generally these tools are used to find new bugs while the regression suite is used to make sure old bugs do not reenter the compiler.

[2]Our compiler development has not been focused on scientific benchmarks so we do not currently include the SPECfp benchmarks in these tests.

SPEC Regress

Our first stress test is composed of all of the SPECint95 benchmarks. Each benchmark is compiled with three different optimization levels and run with seven different input sets, giving us a total of 168 tests. We have found this to be a very difficult test to pass and a number of new bugs were found once we implemented this process. Because of the large amount of time to run these tests we do not run them frequently but rather periodically or when large changes to the compiler source are made.

`brutal`

As part of our regression testing, we test each program in our regression suite built with the sets of optimizations described above. However, a different set of optimizations may expose bugs our regression tests would not catch. We developed the `brutal` tool to run the regression suite with random sets of analysis options and transformation filters. Typically, `brutal` is run overnight and failures are investigated the next day. The programs tested are either chosen by the user or selected randomly. The user can also require that the transformation sequences always include a specific filter. This is used to test new or recently modified filters.

5.2.5 Development Policy

It is important to remember that regression tests or any other validation tool is simply that – a tool. Proper discipline is needed to extract the maximum effectiveness from these tools. We have found that requiring all developers to validate their changes with the regression suite and the larger benchmarks (excluding stress testing) before they commit source changes to the code repository greatly increases productivity. In addition, strong

peer review has allowed us to catch flaws in fundamental module designs before they become major problems later in the software lifespan. While certainly we have not been able to avoid all such flaws, the review process has eliminated a number of them.

No validation tool is a panacea. A regression suite can only detect bugs that have already been exposed. When a new bug enters the compiler, developers often spend days or weeks tracking it down. The next section describes our tools to accelerate and simplify this process.

5.3 Compiler Debugging

Over the course of the compiler tool-chain development, the MIRV team has produced several tools to quickly characterize bugs in the software. The development of these tools was demand-driven and though each can be used independently to aid in debugging, they are most powerful when combined together to provide a systematic approach to compiler debugging.

We define `bug characterization` as the process of locating which parts of the compiler contain the bug (localization) and understanding the cause of the error. Because the compiler is itself generating programs, the characterization process also applies to the compiler output. That is, if a generated program is incorrect, we must determine which parts of the program are buggy and understand why they are producing incorrect results.

5.3.1 Bug Categorization

As we have developed our compiler tool-chain, we have found it useful to group bugs into two broad categories:

a. **Compile-Time**

A compile-time (CT) bug occurs when the compiler stops prematurely for some reason and does not generate a program. Debugging procedures such as setting software breakpoints and examining back-traces often help quickly track down the cause of the bug, though the higher-level information of what sequence of compiler events (filter invocations, for example) triggered the bug is usually more difficult to reconstruct. Examples of situations where these bugs are exposed include unexpected source code constructs, errors in the coding of filters and violation of compiler resource management conventions.

b. **Run-Time**

A run-time (RT) bug occurs when the compiler produces a program, but the program stops early or produces incorrect output for some reason. We assume the program's source is correct, that is, the compiler produces bad code. These are generally much harder to characterize than CT bugs because a debugger cannot trap at the point of incorrect code generation and the bug may not even map directly on to a source code construct (e.g. it is introduced by procedure call conventions or other "hidden" code sequences). Bugs of this type include filters that perform transformations incorrectly and invalid back-end code generation.

5.3.2 Bug Characterization Techniques

The bug characterization tools we have developed also fall into two broad categories:

a. **Source Shrinking**

When a compiler bug is observed, we would like to isolate which piece of code in the input program exposed the bug. This may be a translation unit, a procedure, or even a single line of code. We call this process *source shrinking* because it reduces the

amount of program input code we must examine to understand the bug.

b. **Command Shrinking**

After a bug is exposed we would also like to know what sequence of compiler actions triggered the bug. It is often the case that only a small subset of transformation filters is responsible for the bug and including other transformations in the debugging process simply obscures the real cause of the bug. We would like to reduce the number of options and filter invocations given to the compiler on the command line while maintaining exposure of the bug. We call this technique *command shrinking* because it reduces the number of actions the compiler needs to perform to expose the bug.

5.3.3 Bug Characterization Tools

Over the course of compiler development we have constructed several tools to aid in the debugging process. These tools have proved themselves in production use, providing tangible benefits such as a reduced debug cycle time and batch automation.

`bughunt`

`bughunt` is a source-shrinking bug localization tool that identifies miscompiled translation units. The tool first builds reference object files and MIRV object files for a given program. It then links each individual MIRV-generated object file with the appropriate reference-generated object files to generate an executable. We assume the reference-generated object files are correct and compatible[3] with the MIRV-generated object files. The resulting binary is then tested. If the test fails, it is likely that the MIRV-generated

[3]with respect to procedure call linkage, aggregate object layout, etc.

object file was miscompiled. This process is repeated for each MIRV-generated object file.

The binary incompatibility between MIRV- and reference-generated object files caused problems during our early use of `bughunt`. This was due to differences in how each compiler implemented the UNIX System V Application Binary Interface (ABI), particularly with respect to aggregate type layout [52]. Although both followed the standard, there were enough ambiguities in the ABI to allow for differing interpretations and implementations. To solve this, we modified the MIRV back-end to lay out `struct` objects exactly as gcc does.

`cleaver`

With `bughunt` we are able to quickly localize a bug to a translation unit. Unfortunately, this is usually not fine-grained enough to quickly understand the cause of the bug. Recognizing this, we implemented the `cleaver` tool. `cleaver` takes as its input a MIRV intermediate form file. It is most effective when this file is a source-linked version of the entire input program.

The output produced by `cleaver` is a set of individual MIRV intermediate representation files, one for each procedure in the program. In addition, a set of global declarations is produced and referenced in each procedure file via an `include` directive. Once these files have been generated, the compiler can convert each one to an individual object file. By generating object files for optimized and unoptimized runs of the program, `bughunt` can be used as-is to localize a bug to one or more procedures in the program.

The drawback of this technique is that the buggy compiler is used to generate both the reference (unoptimized) and subject (optimized) versions of the program. The technique will not work if the bug is present even when code transformations are turned off. However,

we have found it to be a very effective tool since once the baseline (non-optimizing) compiler has stabilized, it very rarely changes in ways that expose new bugs at that level. The tool is most useful when introducing new code transformations or increasing the application of existing transformations either through more detailed program analysis or less restrictive constraints.

`auto-cleave`

The `auto-cleave` tool provides the automation link between `cleaver` and `bughunt`. `auto-cleave` uses `cleaver` to break a source-linked intermediate representation file into its component function. It then invokes MIRV to compile each file to a separate object file. It does this for two sets of optimization flags provided by the developer: flags known to produce good code and flags known to produce bad code. After compilation is finished, `bughunt` is invoked to automatically isolate the bug to one or more functions.

By using two nearly identical sets of compiler flags, the developer can get a good idea of what is causing the bug by simply performing a UNIX `diff` between a known-good IR file and a problematic one. Because `auto-cleave` requires the two sets of options to be known in advance, there is some preparatory effort required of the developer. Our later tools address this problem.

LOAR

After `bughunt` and `cleaver` were implemented, we had a fairly powerful tool to characterize compiler bugs. However, we lacked convenience and automation. `bughunt` and `cleaver` require that the programmer manually compile the source program, generate object files for the reference and subject programs, separate those object files into directo-

ries and invoke the tool. We found these steps to be tedious and error-prone, and while `auto-cleave` addresses most of those problems, we still desired something more automatic.

To combat these problems we developed a tool to automatically generate and run new versions of a source program. This tool has the unwieldy name of `lmrvOptimizeAndRun` which we abbreviate as LOAR. LOAR takes a source-linked intermediate form (a "linked MIRV" or lmrv) file, a set of compiler command options and invokes the compiler on the file to produce a new version of the program. The output is another linked MIRV file to which tools such as `cleaver` can be applied. It also generates a program binary and executes it, comparing the output to a reference file. A result code is returned that indicates the nature of the bug (output mismatch, early termination, etc.). While LOAR does not completely address all of the above problems it is a key element of our other tools that do address the above concerns.

`cmdshrinker`

Our first front on the bug characterization battlefield was the problem of generating multiple programs using different analysis and transformation passes. It is much easier to understand the nature of a compiler bug if the number of actions the compiler performs can be reduced. The `cmdshrinker` tool provides this ability. Its input is the name of a linked MIRV file to manipulate and a series of compiler options that are passed verbatim to the compiler.

`cmdshrinker` repeatedly invokes the LOAR tool to compile and execute the program. The LOAR tools reports the nature of any bug found. `cmdshrinker` systematically removes command line options one at a time so that subsequent invocations of LOAR are given a different compiler analysis or transformation sequence or strength. If at any point the return

code from LOAR changes, indicating either the absence of the bug or a different bug, the last command line option removed is placed back into the set of options and the tool proceeds onward, attempting to remove the remaining options. Once all options have been processed the tool outputs the smallest LOAR command line that exposes the bug. It is then a simple matter to apply the UNIX `diff` utility to the two generated intermediate representation files (one with the last command option added and one without it) to characterize the bug. The problem is usually immediately obvious.

TILT

Once a bug is narrowed down to a particular set of filters, the developer would like to determine exactly which code transformation exposes the compiler bug. The Transformation Invocation Limit Tool (TILT) allows the developer to "dial in" the exact transformation that caused the error by using existing compiler command line switches. For example, if the register promotion filter has a bug, we can run with a compiler option such as `-fregPromote=--maxPromotes=N` [49]. Knowledge of the number of promotes in the whole program allows us to determine the exact N's for which register promotion passes and for which it fails. A binary search is employed to find N for which it passes and N+1 for which it fails. The intermediate representations of these two compilations are then compared and the precise promotion that causes the bug is readily evident by using the UNIX diff tool. We can then determine why the promoter thought that the candidate was legal. Of the several promotion bugs that we have found, most of them are due to incorrect alias analysis.

Currently, this is a manual tool as the syntax for limiting transformations varies between filters. The developer simply invokes LOAR with the correct options to restrict the range

of transformation. We plan to standardize this interface to provide an automatic search.

`binsearch`

After using `cmdshrinker` for some time, we realized that it was quite inefficient and often took hours to complete its task. While the automation is certainly beneficial, compiler developers can only play Tetris for so long while waiting for the tool to finish. Clearly, some method to speed up the process was necessary.

The `binsearch` tool was written to solve this problem. Like `cmdshinker`, `binsearch` takes as its input a linked MIRV file to operate upon and a set of compiler options. The tool first examines the input file and determines how many procedures are in the program. Knowing this number, the tool invokes LOAR passing an additional compiler option to restrict analysis and transformation filters to a subset of procedures. The compiler can be told to optimize a particular procedure (by name or number) or a subset of procedures. `binsearch` uses this latter feature to do its work. The tool simply optimizes from the first procedure in the file up to some number and does a binary search to shrink the range to the smallest possible. If the range includes multiple procedures the tool recursively invokes itself to perform a search inside the range to eliminate more procedures from consideration. Multiple procedures may contribute to a bug because inter-procedural analyses and transformations may manipulate some procedure code that affects the correct operation of some other procedure, meaning that restricting the transformations to a smaller subset may hide the bug.

Our first impressions with this tool were very positive. The `cmdshinker` process was accelerated by simply running `binsearch` to reduce the amount of work the compiler had to do on each `cmdshinker` pass. We soon realized that the tools we now had could be

combined to do even greater things.

`findbug`

Given all of the above tools, it is relatively easy for the programmer to track down a bug in the compiler. In fact, we have written several scripts that automate the process of finding bugs by using some combination of the above tools. The `findbug` tool is the most powerful of these.

`findbug` finally addressed our desire for a more automated and easy-to-use bug characterization process. Essentially it is simply a wrapper around `cmdshrinker` and `binsearch`. Given a linked MIRV file to manipulate and a set of compiler options, `findbug` invokes `binsearch` to automatically isolate the bug to a subset of program procedures. In essence, `findbug` uses `binsearch` as a more automated version of the `cleaver-bughunt` combination. We have found that when `binsearch` reports the last procedure it optimized, that procedure is usually the one containing the incorrect code transformation. In the rare cases that an inter-procedural operation has caused an incorrect transformation in some other procedure, we at least know that the bug is exposed in this last procedure and it is usually straightforward to backtrack to the incorrectly compiled procedure.

Once `binsearch` has done its job, `findbug` invokes `cmdshinker` with the compiler options given plus the option to restrict transformation to the subset of procedures `binsearch` found. `cmdshinker` is told not to eliminate this last option during its search. Once `cmdshrinker` is done the developer can use the `diff` tool as described above to quickly track down the problem. In the rare case where `diff` does not provide enough information, the developer can use the Transformation Invocation Limit Tool to reduce the number of diffs generated.

5.3.4 Tool Categorization

Both bug-search approaches, source-shrinking and command-shrinking, can be used to find bugs, whether they are CT or RT bugs. Table 5.1 categorizes each of our bug hunting tools. Most of the tools were written to use a specific technique to isolate bugs. The tool that uses both techniques, `findbug` was written specifically to do so and unifies our more advanced tools to leverage the benefits of each technique. Most tools address both types of compiler bugs. `bughunt` operates on object files so it cannot isolate compiler-time bugs and TILT operates manually as part of the compilation process. While it can be used to characterize compile-time bugs we have found it easier to simply run the compiler in a debugger since we are invoking it manually anyway.

Tool	**Category**	**Type of Bug Isolated**
`bughunt`	Source Shrinking	RT
`cleaver`	Source Shrinking	CT, RT
`cmdshrinker`	Command Shrinking	CT, RT
TILT	Command Shrinking	RT
`binsearch`	Source Shrinking	CT, RT
`findbug`	Source, Command Shrinking	CT, RT

Table 5.1: Bug Isolation Tool Categorization

Figure 5.1 shows the debugging process used with the MIRV compiler. Solid block arrows represent automatic tool invocations, solid grey arrows represent information passed back to a tool and dashed arrows represent developer input and invocation. The lower-left corner represents the regression and stress testing procedure. The upper-left corner group shows the `cleaver`/`bughunt` interaction to source-shrink a bug. The rightmost group shows the `findbug` tool in action. This is the most automated process of the three and can apply both source and command shrinking to systematically and automatically characterize a bug.

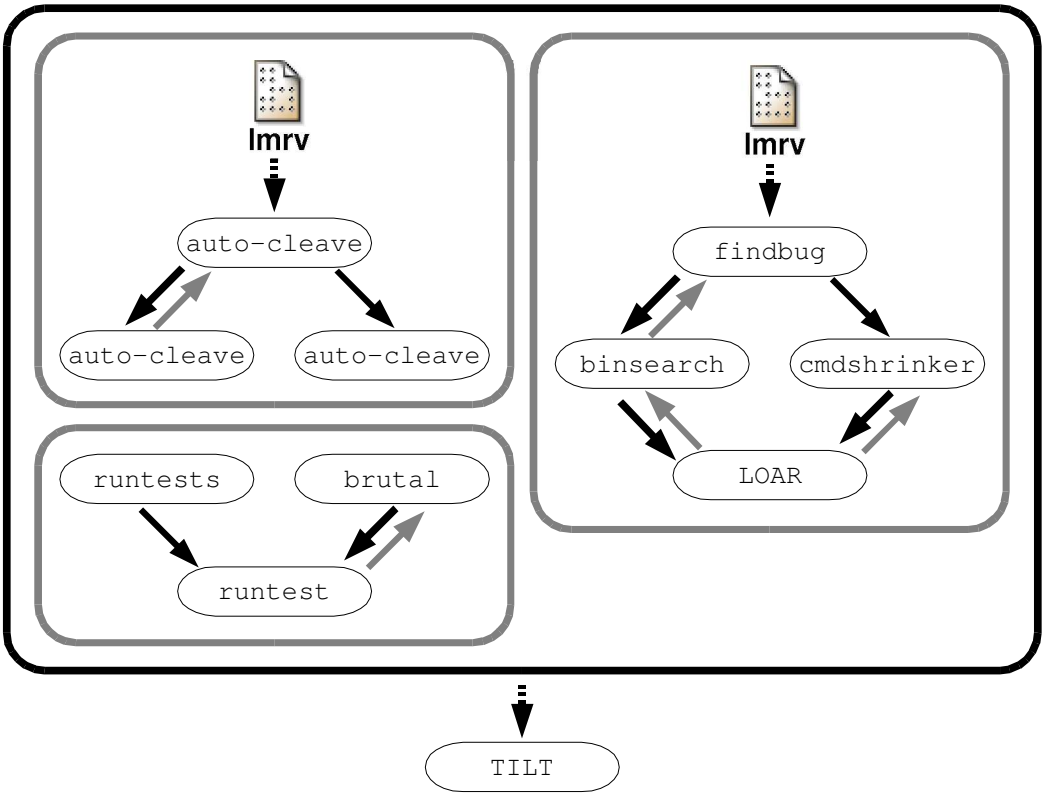


Figure 5.1: MIRV Debug Flow

5.3.5 The MIRV Architecture

Chapter 2 provides an overview of the MIRV architecture and its operation. We briefly note here how this design greatly aided the development of these debugging tools.

As noted above, the LOAR tool operates on MIRV intermediate representation files and is most effective when these files are source-linked representations of the whole program being compiled. It is this source-linking that allows `binsearch` to do its work automatically. Certainly such a tool could be written to operate in a separate compilation environment but it would require the maintenance of some sort of external database to track its progress. Given a linked MIRV file, the tool can immediately know the size of the program and the necessary compiler invocations to limit its operation to a subset of that code.

The automatic operation of `binsearch` and `cmdshrinker` is necessary for a tool like `findbug`, whose great power lies in automation. Automation of the bug characterization process allows the developer to work on other projects while the bug is being found. MIRV's source-linked representation allowed us to quickly develop and deploy these tools in a production environment.

5.4 Experiment Methodology

One of the goals of the MIRV compiler project is to provide an easily accessible set of tools for researchers in compiler development and computer architecture. The design of MIRV itself allows relatively independent development of analysis and transformation filters along with experiments in back-end code generation for novel architectures.

Software architecture is only one aspect of "easily accessible," however. Correctness is of equal importance. Our goal has been to provide the same level of confidence with our

tools that researchers have with the tools currently available. Because we have focused our research on the SimpleScalar/PISA and IA32 architectures, we chose the GNU gcc compiler as our confidence target.

5.4.1 Other Compiler Toolchains

In this study we explore how MIRV performs against the gcc compiler available for SimpleScalar/PISA when running our regression suite. In addition, we analyze two other popular research compilers. In general, we make no claim about the relative quality of any tool-chain for a particular task – often these tools were developed with specific target audiences in mind and it is important to remember that outside of those domains problems are to be expected. It is quite likely that MIRV would not perform as well as these other tools on their own regression suites.

Our hope is that through the presentation of these results we can encourage researchers to critically analyze their tools and improve the quality of all such software. Additionally, we wish to make the case that solid research compiler tools, often stated as a high-priority item in the research community, are extremely difficult and time consuming to develop, with the unfortunate conclusion that such tools are currently few and far between.

To provide a consistent test across multiple compilers, we ran our regression tests on each system using the lowest possible level of optimization. We describe these options in more detail below. Since MIRV currently passes all of the tests run on these other compilers, the set of options passed to it is irrelevant for this experiment.

gcc

Our first test subject is the SimpleScalar/PISA port of gcc done by researchers at the University of California, Davis. This port is based on gcc version 2.7.2.3. This particular port, being a derivative of the existing MIPS port, does not make use of all PISA features, particularly the register+register addressing mode. MIRV does make use of this mode. For this test we run gcc with O0 optimizations. At this level gcc simply generates code and registers are used only as temporary storage.

`lcc`

Our second tool is the `lcc` retargetable C compiler developed at Princeton University [51]. `lcc` was designed to be ultra-portable and consequently does not support program optimization. Because there is not PISA target for `lcc`, we ran the IA32 version of the compiler. As optimization is not supported, `lcc` simply allocates registers and generates code.

MachSUIF

Our final candidate is the MachSUIF compiler. SUIF is a source-to-source parallelizing optimizing C compiler developed at Stanford University [26]. MachSUIF extends the compiler by providing an optimizing machine-dependent backend to generate executables [29]. We used the IA32 target of MachSUIF version 2.00.12.12 based on SUIF 2.2.0. Both SUIF and MachSUIF are part of the ongoing National Compiler Infrastructure project. For this test we used the minimal sequence of MachSUIF program transformations which includes register allocation as the only optimization.

Failure Type	Description
alloca	Doesn't support `alloca`
bit-field	Doesn't support all allowed types for bit-fields
goto	Problem with `goto` label position
init	Problem with constant static variable initialization
cast	Rounds instead of truncating when casting floats to integers
pack	Problem with aggregate type packing
reg alloc	Failure in the register allocator
setjmp	`setjmp` or `longjmp` failure, generated program never exits
call	Failure in procedure call linkage generation
codegen	Generates illegal machine code
misc	Problem wasn't diagnosed

Table 5.2: Failure Category Descriptions

5.5 Results

In this section we report the results of the experiments described in section 5.4. In addition to raw numbers, we also attempt to categorize the failures in order to better understand the improvements that can be made to the compiler tools available to the computer architecture research community.

5.5.1 General Results

Overall, all the compilers performed well on our regression suite. Out of the greater than 600 tests run, `lcc` failed 15 of them and MachSUIF failed 36. In addition, gcc failed one test[4]. This was quite a surprise for us because it is a very mature compiler and the PISA ISA and ABI is nearly identical to the MIPS variants. If the compilers were run with optimizations enabled these numbers could very well rise dramatically[5].

Below we categorize the results in an attempt to characterize compiler deficiencies. Table 5.2 explains the failure categorizations.

[4]Since gcc is our correctness standard, we attributed this particular failure to gcc after close examination of the output produced by MIRV and gcc.

[5]Of course, `lcc` does not perform transformations so its failure count should not increase.

benchmark	alloca	bit-field	goto	init	cast	pack	misc
alloca.c	X						
allocaarg.c	X						
bitfieldeasy.c		X					
castfloatbyte.c					X		
castfloatint.c					X		
castfloatshort.c					X		
castintfloatdiv.c				X			
gcc-reload1.c		X					
gotofwdunstruct.c			X				
ptrarithinit.c				X			
scribpack.c						X	
shift.c							X
ss.test-math.c							X
strlen.c							X

Table 5.3: lcc Failure Categorization

gcc

As would be expected, gcc performs very well on our regression suite. This makes sense because the suite was developed using gcc as the reference compiler. gcc fails only one test: `lcc.stdarg.c`. It seems that the PISA port of gcc does not completely implement the PISA calling convention correctly.

`lcc`

The `lcc` results presented in table 5.3 show that it suffers from a mixture compile-time and run-time problems. On the compile-time side it mainly seems to suffer from incompleteness. It does not handle the full range of possible bitfield types. There are also restrictions on the placement of `goto` target labels. In some cases it does not allow the label to appear at end of a statement block, or a block that isn't otherwise reached. It also doesn't fully support global variable initialization with complex constant expressions, in particular complex pointer arithmetic. However, in all these cases the compiler generated a reasonable error message. This is a testament to the maturity of the project.

We discovered various run-time bugs. That is, the compiler generates programs that execute but produce invalid output. The most frequent problem involved the rounding of floating point numbers when converted to integral types. The ANSI C standard calls for truncation [21]. It also doesn't follow the ANSI standard or UNIX System V ABI with respect to aggregate object layout. In all these cases, benchmarks specifically designed to catch these problems are in the regression suite.

MachSUIF

The MachSUIF results presented in table 5.4 show that it also suffers from a mixture of compile-time and run-time problems. On the compile-time side there are problems with register allocation, stack setup for procedure invocation and `gotos`/labels in a switch statement[6]. Unlike `lcc`, SUIF didn't generate any type of error message, instead in all these cases the compiler terminated upon hitting an assert statement and simply returned the compiler source file and line number. One of the tests caused a memory access fault in the compiler.

On the run-time side the most common problem by far was the generation of illegal assembly code. The compiler generated code that contained invalid or completely missing operands. For one benchmark, the compiler didn't report any errors, but the generated assembly code was only 10 lines and simply contained a few symbol declarations, with no executable code generated. `setjmp` and `longjmp` are completely unsupported by this version of MachSUIF, as it generated executables that did not terminate.

[6]such as one finds in Duff's Device

5.5.2 A Case Study: newlib

Because we desire to create a complete compilation environment for computer architecture researchers, it is necessary to provide a standard library along with the compiler. Currently we use a binary version of GNU `glibc`. We cannot ship a source version because `glibc` is heavily dependent on compiler extensions present only in gcc. Therefore, part of our ongoing work involves porting another C library to our system. We have chosen the newlib C library for this task [53].

newlib is a portable C library originally developed for use in embedded systems. It is not tied to any particular compiler, which makes it ideal for our situation. We chose to start our porting effort with version 1.10.0 of newlib, the most current as of this writing.

Our original plan for the process was to begin with a retarget of newlib for the PISA ISA and SimpleScalar simulator using the PISA version of gcc. Once this was complete, we could then work on compiling newlib with MIRV. However, the PISA gcc could not compile several of the math library files when using O2 or greater optimization. The compiler terminated with an uncaught error[7] indicating inconsistent intermediate code generation[8]. The files in question contain implementations of the `asin` and `log1p` routines. The code is pure C floating point code with no target specific dependencies. We reduced the problem down to a single source file and MIRV was able to compile it at full optimizations without problem. The IA32 version of gcc could also compile it.

As the PISA port of gcc is an older version of the compiler, it's quite likely that a port based on a more recent version of the gcc system would handle the files in question. However, no such port currently exists. We have not yet had the opportunity to test compilation with

[7] "Internal Compiler Error" in gcc parlance

[8] The compiler indicated that it generated an instruction that did not satisfy its own input operand constraints.

other available tools. We did find it interesting, however, that MIRV performed well on code that it had never encountered before while gcc, a system with over a decade of development behind it, failed to compile a file produced by many of the same people who have spent time developing gcc itself. We believe this points to the effectiveness of our regression and debugging tools.

5.6 Previous Work

The task of fault detection, characterization and correction has long been recognized as a critical path in the software development and maintenance cycle. A wide variety of techniques have been proposed to reduce the cost of this process.

Program Slicing can be used to ease the validation of changes to existing software [54]. The program is analyzed to determine the portions relevant to a particular value computed. Changes are only allowed to affect this subset of the program. Because side-effects from the changes are eliminated, full regression testing need not be performed on the modified program. In essence, program slicing is used to create a smaller version of the program that only computes the essential state needed for the portion of the program being modified. This technique could be applied to compiler input programs to aid automatic regression test generation.

The expense of regression testing can become prohibitive in some cases. To combat this problem regression test selection techniques have been developed to reduce the number of tests that must be run to validate a change [55, 56]. Selection assumes that tests have been associated with parts of the program and provide adequate coverage. The selection process involves finding a subset of these tests that cover the changed portions of the program.

Algorithmic Debugging is a methodology used to reduce the time to locate the source

of a program bug [57]. Once a bug has manifested itself the debugger walks through an execution trace of the faulting program and asks the user a series of questions about the expected state of the execution. Algorithmic Debugging could be used in conjunction with the tools presented here. Our tools can locate where a compiler bug manifests itself in the output program through source shrinking. Command shrinking is able to locate the bug in the compiler in a very coarse-grained manner. Algorithmic Debugging can be used to narrow this focus. We note that the questions asked by the algorithmic debugger are likely to be quite complex due to the amount of program analysis state gathered by the compiler.

Algorithmic Debugging research has focused on providing a semi-automatic bug isolation system applicable to general software development. Our tools were developed within a specific domain: compiler construction. Because of the additional context available, we have been able to nearly completely automate the bug characterization process. After the tools have been run there is still some programmer effort involved in pinpointing exactly which piece of compiler code is buggy, but the tools give a very good idea of which parts of the compiler to examine. Most often it is immediately obvious which transformation filter is buggy. Frequently the cause is incorrect dataflow analysis and there may be a non-trivial amount of effort to backtrack to this culprit. Algorithmic Debugging can help with this process.

While the above techniques aim to quickly find and eliminate bugs, Software Fault Isolation is a technique to tolerate program bugs to some degree [58]. In this work program modules are altered so that individual memory operations are guaranteed to stay within a certain region of memory. This allows the modules to share a single address space with the guarantee that a fault in one module will not affect another.

5.7 Conclusion

We have presented a set of tools to aid experimental research compiler developers. In addition to the standard regression testing techniques, the tools provide a framework for automatic bug detection and characterization, reducing the burden of the developer. Through the use of source and command shrinking techniques, the tools rapidly localize a bug to short input code and transformation sequences.

The architecture of the MIRV compiler aided the rapid development of these tools. In particular, the source-level linking capabilities of the compiler removed the burden of maintaining external databases or developing binary manipulation tools with the associated code duplication costs. In addition, exposure of the compiler operation through a wealth of command-line options increases the effectiveness of the command shrinking technique. Without the fine-grained control available in MIRV, tools such as `cmdshrinker` loose much of their power. Finally, we also note the filter dependency structure used by MIRV. Because filters declare the attributes they are dependent upon, the developer need not worry about filter ordering when writing tools such as `brutal` that reorder the filter passes.

All of the tools presented in this paper have been in production use for quite some time. They have proven themselves invaluable to the small team of researchers developing the compiler software. MIRV compares favorably with respect to correctness to compiler systems that have been available for much longer periods of time and with much larger development teams. All of the work on MIRV was conducted with no more than three developers on the team at any one time[9]. We attribute this to the multiplicative effect of the tools on developer productivity.

We believe there is much room for additional tool development. In particular, we are

[9] frequently, only two!

working on tools to automatically generate regression test cases once our tools have isolated a bug. This is a much more difficult problem as the tool must maintain the semantic requirements of the environment that exposed the bug. In the future we hope to reach a point where the bug detection/isolation/test creation cycle can be fully automated.

benchmark	alloca	reg alloc	setjmp	call	goto	codegen	misc
alias6.c		X					
aliastest.c						X	
alloca.c	X						
allocaarg.c	X						
bitfieldeasy.c						X	
callbyte.c						X	
castintbytecall.c						X	
colorme.c		X					
constprop-setjmp.c			X				
copyProp-setjmp.c			X				
divide.c						X	
doomlevels.c						X	
floatbool.c						X	
floatzero.c						X	
indcallstructret.c				X			
lcc.cvt.c						X	
lcc.fields.c							X
lcc.incr.c						X	
lcc.struct.c							X
lcc.switch.c							X
lhsfun.c				X			
localoffsetshift.c				X			
longjmp.c			X				
putbyte.c						X	
regs.c		X					
regs34.c		X					
ss.anagram.c							X
structcpy.c				X			
structreturn.c				X			
structreturn2.c				X			
switchWithGotos.c					X		
switchWithGotos2.c					X		
ternary3.c						X	
ternaryvoid.c							X
unionreturn.c				X			

Table 5.4: MachSUIF Failure Categorization

CHAPTER 6

Instruction Prefetching

6.1 Introduction

The increasing gap between memory and processor core performance requires that modern computer system designs either eliminate or tolerate the increasingly large penalty of accessing lower levels of the memory hierarchy. In particular, fetching instructions efficiently from the memory system is critical, as the front-end capacity of the machine's pipeline puts a cap on its overall processing bandwidth. One study of commercial database and web applications shows that as many stall cycles are dedicated to servicing instruction cache misses as data cache misses, sometimes approaching 50% of all stall cycles [59]. This paper concentrates on studies of instruction fetch efficiency.

Memory latency can be eliminated by placing data in small structures that provide fast access time. Such structures include upper-level memory caches and register files. Alternatively, prefetching may be used to tolerate the latency by requesting the desired data in advance of when it will actually be used. By properly timing such requests, the machine can reduce the number of cycles spent waiting for the data, or eliminate them all together.

Several instruction prefetching techniques have been proposed in the literature. Initially, most of these techniques were implemented entirely in hardware. Recently, techniques for software instruction prefetching have been proposed. There are tradeoffs involved in each approach. At run-time the hardware can make use of contextual information such as cache miss points, branch history or address stream history to predict when a prefetch may be useful [60, 61, 62, 63]. At compile-time, the compiler can obtain a wider view of program structure to predict likely instruction miss points in the program [45, 64]. The software-based techniques have the additional advantage of reducing hardware complexity and cost at the expense of compiler complexity and slightly increased compile times.

Most of the recent studies of instruction prefetching concern themselves with evaluating proposed techniques on aggressive or "near-term" high-performance microprocessor designs. Typically, these designs have out-of-order cores with large issue widths. Such machines are designed to tolerate memory latency by overlapping cache misses with useful instruction execution. While this is primarily targeted at tolerating data cache misses, some amount of instruction miss latency can be overlapped given a sufficiently large (and full) instruction window.

In this chapter we evaluate various instruction prefetching techniques on several machine organizations. We examine the utility of instruction prefetching schemes on both current and "near-term" processor designs. Due to their relatively recent appearance, we choose to examine various software instruction prefetching techniques to determine their utility on these designs. Much of these studies focus on areas that have been left underspecified by previous work in instruction prefetching, such as the hardware mechanisms to schedule prefetch requests and the associated timing constraints. In addition, we propose various extensions to the software algorithms and hardware designs to increase the effectiveness of

software instruction prefetching.

The chapter is organized as follows: section 6.2 discusses the existing techniques we study and summarizes their operation. Section 6.3 discusses various architectural considerations for software instruction prefetching which have been underspecified in previous work. We discuss design alternatives and tradeoffs. Various software algorithms are discussed and generalized in section 6.4. In section 6.5 we explain our experimental methodology, including software algorithm implementations, machine models and experiment design. Section 6.6 contains an evaluation of the existing techniques on various machine organizations. We discuss and compare our findings with the existing literature in section 6.7 and conclude in section 6.8

6.2 Overview

This section presents a brief overview of the baseline instruction prefetching algorithms we wish to examine. Purely hardware solutions are presented first and the more recent software techniques are then considered. The instruction prefetching literature is very rich and examining all such algorithms proposed is beyond the scope of this work. Therefore, we have elected to examine techniques that are simple and relatively cheap to implement in hardware. Section 6.7 provides a wider sampling of existing techniques.

6.2.1 Sequential Prefetching

One of the earliest instruction prefetching techniques studied is the venerable sequential prefetcher [65, 60]. Sequential prefetching is a highly effective technique because program execution usually proceeds in a sequential manner or if branches are encountered, they are often short. The main drawback of sequential prefetching has been timeliness. Because

prefetching is to the next N cache blocks, they may not be enough time to hide the latency for a cache miss, especially if the current access is a cache hit.

6.2.2 Branch History Guided Prefetching

To combat the timeliness problem, Srinivasan, *et al.* proposed *Branch History Guided Prefetching* (BHGP). In this technique, the machine maintains a queue of the last N branches encountered. On a cache miss, the culprit address is associated with the branch at the head of the queue (i.e. the Nth previous branch). When that branch is encountered again a prefetch is initiated to the missing address. The branch queue ensures that prefetches are initiated earlier than they would be in a sequential prefetching scheme.

Recently, software instruction prefetching has been proposed as a technique to improve the timeliness of instruction prefetches. Some researchers have argued that the compiler can better schedule prefetches far enough in advance to cover the latency of a cache miss. In this work we examine various software prefetching alternatives, both in the abstract sense of algorithm design and in practical application. The first technique is *Call Graph Prefetching*, proposed by Annavaram, *et al.* [64]. The more general *Cooperative Prefetching* proposed by Luk and Mowry [45] follows. These algorithms form the baseline for some new algorithms proposed in this work: *Compiler Hint Guided Prefetching* (CHGP) and *Cooperative Compiler Hint Guided Prefetching* (Cooperative CHGP).

6.2.3 Call Graph Prefetching

Call Graph Prefetching (CGP) attempts to take advantage of the wide scope of compiler program analysis and transformation by inserting instruction to prefetch function call targets. Annavaram, *et al.* study both hardware and software implementations of CGP

```
CollectDynamicCallGraph()
for (each function call)
   insert prefetch for next function after call
InsertSequentialPrefetches()
```

Figure 6.1: Software CGP Algorithm

[63, 64]. As we are primarily interested in software prefetching algorithms, we only consider Software CGP in this study.

Software CGP operates in two phases: the first constructs a static call graph given a program binary. An instrumented version of the binary is run and the resulting profile information is used to label the static call graph with information about the order of function invocation. In the second phase, a binary rewrite tool is used to insert prefetches into the existing binary. Prefetches are inserted for each call in its dynamic sequence. For example, if procedure A calls procedures B, C and D in that order, CGP will insert a prefetch for B at the top of A, then insert a prefetch for C immediately after the call to B, and so on for all callee functions. Thus the run-time profile acts as a static prefetch filter by eliminating prefetches for functions that were not invoked during the profile run.

The CGP prefetch instructions are able to prefetch N lines at a time (N=4 in the study by Annavaram, *et al.*). In addition to these inter-procedural prefetches CGP inserts prefetches for N cache lines at equidistant intervals throughout each function body. These instructions attempt to emulate in software the operation of a hardware next-N-line prefetcher [60]. These software next-N-line prefetches also attenuate the bandwidth and pollution problems of prefetching large functions into the instruction cache. The inter-procedural prefetches only prefetch the first N lines of a callee function. The remaining lines are only prefetched if execution in the callee passes through the software next-N-line prefetch instructions.

```
  SchedulProfetches(B: basic block,
                    T: target block,
                    D: prefetch distance between T and B)
  for(each block B in function) {
A:  if (have not considered B for target T) {
      MarkConsidered(B, T);
B:    prefetched = HardwarePrefetched(B, T) || SoftwarePrefetched(B, T)
      if (!prefetched) {
        if (D >= targetPrefetchDistance && !LocalityLikely(B, T)) {
          InsertPrefetch(B, T);
          prefetched = true;
        }
      }
      if (!prefetched) {
        for(each predecessor block P of B) {
          newDistance = D + CountInstructions(P);
          SchedulePrefetches(P, T, newDistance);
        }
      }
    }
  }
```

Figure 6.2: Cooperative Prefetching Algorithm

Figure 6.1 presents pseudo-code for the software CGP algorithm. The algorithm is very simple, simply inserting prefetch instructions after each function call in the program to prefetch the next mostly likely function call target. As noted above, sequential software prefetches are inserted at equidistant points throughout each procedure.

6.2.4 Cooperative Prefetching

Cooperative Prefetching attempts to provide a bridge between hardware and software prefetching schemes. Software prefetch instructions are used to prefetch over large breaks in control flow (e.g. over a function invocation or distant-target branch). A hardware next-N-line prefetcher covers the cache misses over shorter-distance control flow structures (e.g. sequential or near-target branches).

The basic algorithm (shown in figure 6.2 and simplified a bit from the original presentation by Luk and Mowry) is quite simple. For each basic block B in a function, the compiler walks backward through the control flow graph of the program until a specified distance N is reached (N=20 in the study by Luk and Mowry). At that point a software prefetch targeting B is inserted at the top of the current block. The walk back through the control flow graph ensures that all paths to B are covered by software prefetches. In addition, targets of any function calls are prefetched in a similar manner. Indirect jumps (calls and branches) are prefetched using a software prefetch instruction that queries a hardware structure to produce multiple prefetch targets. Luk and Mowry conclude that such instructions improve performance minimally so we ignore them in this study.

There are a few important points to note about this algorithm. During the walk, any function calls encountered contribute a distance factor equal to the shortest dynamic path through the callee function, taking into account the shortest paths of functions the callee calls. Because the compiler can only analyze code statically, some heuristics are used to guide this computation, such as assuming that each loop body is executed at least once. The `LocalityLikely` function determines whether a target block T and a block B which is being considered to hold a prefetch for T are both in the body of a small loop, where "small" is defined so that the loop body can fit into the cache. This is a simple static filtering mechanism built into the algorithm. Likewise, `HardwarePrefetched` and `SoftwarePrefetched` are static filters that eliminate some redundant prefetches. We extend this filtering concept in section 5.4 where we morph this algorithm into a more generic form.

To prevent a glut of software prefetch instructions from saturating the instruction fetch engine and memory subsystem, the compiler implements various prefetch filters and opti-

mizations. These include combining prefetches at dominator blocks [1]), removing prefetches covered by other software or hardware prefetch operations and compressing sequential prefetches into single multi-target software prefetches. In addition, a confidence-based hardware filter is used to reduce prefetch memory traffic even further by squashing dynamic prefetch operations that are determined to be ineffective.

Unfortunately, early studies in the course of this research have shown these filters to be extremely sensitive to the procedure size estimation heuristics. While Luk Mowry comment that a procedure of size 1000 instructions is "large," we have found the recursive algorithms size estimation algorithms presented in their work to easily estimate sizes on the order of millions of instructions. Such large sizes often prevent the movement of prefetches into dominator blocks because such placement is deemed too likely to interfere with the caching of code in-between the prefetch and its target line. The large discrepancy in size estimation may be due to the compilation model employed. Our compiler sorts procedures in the program in a reverse topological ordering based on the static call graph. This way sizes for leaf functions are estimated before their callers so that there are fewer unknown procedure sizes when procedures later in the sort are estimated. Because their studies employ a post-link pass it is likely that such a sort is not performed on their benchmarks programs.

Cooperative Prefetching is more general than Call Graph Prefetching because the software instructions target more than function call targets. Even so, Luk and Mowry show that the inter-procedural instruction prefetches account for most of the gain seen by Cooperative Prefetching over a purely hardware-based sequential prefetcher. Cooperative Prefetching also has the advantage of using full compiler knowledge to schedule its prefetch instructions. Because Call Graph Prefetching uses profile information, it may miss some prefetch opportunities. On the other hand, the purely static nature of the Cooperative Prefetching

compiler algorithm drives placement of more prefetch instructions than are necessary to cover the important dynamic misses. Luk and Mowry apply profile information to remove some of these useless prefetches and show little degradation in performance of the technique.

In section 6.4 we present a generalized algorithm that covers both the Cooperative Prefetching and Software CGP algorithms and provides enough extensibility to allow expression of other prefetching algorithms. In addition, we describe a general framework for characterizing software instruction prefetching algorithms.

6.2.5 Compiler Hint Guided Prefetching

The major drawback of the above software techniques is the overhead associated with inserting prefetch instructions. Such instructions add to the cache footprint of the program and can offset the advantage gained by instruction prefetching. The primary difficulty is that the compiler has very little knowledge about the miss behavior of the program. Heuristics exist that attempt to gauge whether a piece of code will likely be in the cache or not but our experience is that such heuristics are extremely sensitive and often do not reflect reality.

These discoveries led us to consider two additional prefetching techniques. At run-time the machine has a very good view of the miss behavior. However, the compiler has a very good static view of the scheduling requirements for potential prefetch targets. Therefore, these techniques combine aspects of other software and hardware techniques.

On the software side, we follow the lead of Luk and Mowry to schedule instructions. We use a generalized prefetch scheduling algorithm presented in section 6.4 but instead of inserting prefetches we mark existing instructions with a hint for the hardware[1]. At run-

[1]We assume that enough opcode space is available to make this possible.

time a prefetching table similar to that used in BHGP associates instruction cache miss addresses with instructions that have their hint bits set. When a marked instruction is encountered, it queries the address table. If an association is found a prefetch of length four is initiated to the target address. We arbitrarily chose distance four because that is the prefetch distance used by the prefetch instructions in Cooperative Prefetching. The instruction then checks whether a cache miss has previously has occurred. If so, it associates the miss address with the address of last previously seen marked instruction. The address of the current marked instruction is then saved in a register. In essence we replace the branch queue of BHGP with the scheduling algorithm of Cooperative Prefetching. Our hope is that the software algorithm can improve the prefetch timeliness as BHGP is entirely at the mercy of the program basic block size, which can vary widely.

We call the above scheme *Compiler Hint Guided Prefetching* (CHGP) and propose two variants. The first follows the general operation of BHGP in that the hardware prefetcher is responsible for all prefetching operations. Our second variant, which we call *Cooperative CHGP* maintains a sequential prefetcher for covering sequential and short branch accesses, a technique with proven effectiveness. The software algorithm includes filters to prevent marking of instructions that are covered within the scope of the sequential prefetcher, leaving the prefetch table free to cover only the long-distance prefetches. Both techniques are in fact "cooperative" in that a software algorithm provides support to a hardware prefetching mechanism. Our titling distinguishes between techniques that use a single prefetching mechanism and techniques that employ multiple "multilateral" prefetchers.

6.3 Prefetch Architecture

The plethora of hardware and software prefetching algorithms presents a complex problem for the architectural designer. Because prefetches may be issued under widely different circumstances and with highly variable timing, it is important to systematically examine the possible variations between hardware prefetching implementations.

In this section we discuss points within the architectural design of instruction prefetchers that may present a variety of design choices. We begin with an overview of the literature and show that a variety of often unspecified assumptions about the underlying machine architectures have been made. We categorize these variation points and explore a variety of design alternatives for each. In section 6.6, we present a set of experimental results to quantify the effects of each of these design decisions.

6.3.1 Literature Overview

In this section we examine some of the existing instruction prefetching literature and explain design points that are underspecified. This ambiguity makes reproduction of published results problematic at best. One of the goals of this study is to enumerate as much as possible those design areas which may impact prefetching performance and suggest viable alternatives to maximize effectiveness. We call the design variations *policies* for instruction prefetching.

One of the first questions that arises when designing an instruction prefetcher is when to initiate prefetching. We call this the *initiation policy*. The two most obvious choices are to prefetch only on a cache miss or to prefetch on every cache reference. These two policies were examined in the context of a one-line sequential prefetcher by Smith [65]. He concluded that prefetching on every reference gave the best performance.

Smith and Hsu explored one-line sequential prefetching in the context of pipelined supercomputers [60]. It is not clear whether their prefetcher was triggered on a miss or any reference. The only initiation policies they examined were the distance before the end of a line access at which to initiate prefetching and the use of prediction tables to initiate prefetching for non-sequential accesses.

To our knowledge, no studies have examined prefetch initiation in the context of aggressive multi-line sequential prefetchers. It is not clear whether prefetching on every reference will pollute the cache with distant prefetches that are never used. Our studies will clarify this point.

When prefetching long sequences of instructions, one must decide when to stop the prefetching so as not to pollute the cache with useless prefetches. Xia and Torrellas proposed an extension to sequential prefetching in which the compiler marks the end of a sequential prefetch sequence [66]. The hardware prefetcher can run far ahead of the fetch engine and prefetch. The stop markers cause the prefetch engine to terminate, reducing the number of useless prefetches generated.

The sequential prefetcher is quite smart in that a demand miss to the cache causes it to terminate the current prefetch sequence and start a new one at the miss address. In addition to the sequential prefetcher a software scheme is used to prefetch across long-distance branches.

For our purposes, we wish to examine the utility of the smart sequential prefetcher. We model the baseline and smart sequential prefetchers by describing how a prefetch sequence is terminated. The baseline prefetcher simply prefetches until the desired prefetch distance is covered. The smart sequential prefetcher will always terminate (redirect) a prefetch sequence on a cache miss. Because Xia and Torellas do not model any sort of prefetch

request buffer, it is not clear what happens to prefetches from the old sequence that may still be pending in the buffer. We model this design point with a *prefetch termination* policy.

The Cooperative Prefetching work of Luk and Mowry is similar to the work of Xia and Torrellas but lacks the smart sequential prefetcher. To compensate, their software algorithm is more complex and uses various heuristics to schedule software instruction prefetches effectively. As in the other work, the hardware interface to the cache is underspecified. In particular, no prefetch request buffer is identified and the number of ports available to check for in-cache prefetch targets is unspecified. The last point is crucial to understanding why a request buffer is necessary. If the number of ports on the cache is limited then it is possible that prefetch addresses for the multiple-line prefetches generated by the sequential prefetch hardware and software prefetch instructions will be generated more quickly than the hardware can check the cache for in-cache addresses. The buffer decouples target address generation from cache access.

Neither of the previous work on Call Graph Prefetching and Branch History Guided Prefetching specifies the cache interface in any great detail. At best a one-cycle delay for accessing hardware tables is enforced. It is not clear when branches are processed in BHGP. One option is to use the outcome of the branch predictor to determine whether an instruction address should be sent to the prefetcher. The predictor indicates whether the instruction *may* be a branch. Alternatively, we may wait until decode to guarantee that the instruction is indeed a branch. We must also clarify another point. BHGP uses branch targets to begin prefetch sequences. The target could either be the predicted target of the branch or the actual target produced after decode. We use the predicted target because it is where fetch was directed after the branch was encountered. Srinivasan, *et al.* specify using "the address of the instruction that followed the most recently executed branch." This

could be interpreted as the address produced by the most recent branch exiting decode, the address produced by the most recent branch exiting execute or the predicted target of the most recently fetched branch.

6.3.2 Design Variation Points

Given the above brief literature survey we can identify several points of potential variation in prefetch hardware design. These can be classified into seven major categories. The first task of the engineer is to design the interface between the processor and the cache. Usually this involves some sort of queueing structure and possibly additional logic to generate multiple prefetch requests. The second variation point determines when prefetch requests are generated. The third concerns the replacement policy used by table-based hardware prefetchers and is not applicable to all designs. The fourth concerns generation of multiple prefetch requests from a single event such as a cache miss or software prefetch instruction. A fourth category defines how multiple prefetch request sequences are terminated. The sixth category concerns the scheduling of prefetch requests to the cache, especially in cases of multilateral prefetching. Finally, the cache itself must decide how to prioritize requests. Are demand misses more critical than prefetches? Or is it more beneficial to schedule prefetches early in order to maintain timeliness?

Cache Interface

To support prefetching, some hardware is needed to interface between the processor core and the cache in order to generate and/or queue prefetch requests. Designs here may very widely and we only consider two main architectures. The first is a simple FIFO queue that accepts prefetch addresses and sends them to the cache on a first-come, first-served

basis. Attempts are made to merge duplicate requests in that new prefetch sequences will only generate requests that are not already in the queue. No attempt is made to track previous prefetch sequences in order to additionally reduce the number of requests generated and cache bandwidth used to service those requests. For example, if we assume eight-line sequential prefetching, a demand fetch to line zero will trigger prefetches line lines one through eight. Lines one through three may be sent to the cache in the same cycle as the demand fetch assuming the additional policies explained below allow it. The next cycle, line one may be fetched which will trigger prefetching of lines two, three and nine, overlapping requests generated previously but already sent to the cache.

Our other design follows, but does not duplicate exactly, that of Xia and Torelles [66]. This queue includes state to keep track of the current sequential prefetch path being generated. Further requests along this path will be shortened so as not to overlap prefetches that have already been issued to the cache. For example, if we assume the queue is in the initial state, a demand fetch to line zero will trigger prefetches to lines one through eight. Demand fetch of line one will only trigger a prefetch of line nine. With this design the prefetch sequence termination policies discussed below come into play.

In addition to queue design, other factors must be considered. Most prefetching schemes include various filters to reduce the number of useless prefetches sent to the cache. An obvious filter checks the cache to identify prefetch targets that are already present. Such checks necessarily use cache ports and this resource utilization is modeled.

Prefetch Initiation

Our next design issue concerns the policy of prefetch initiation. As the processor is executing a program, various events can be used to trigger instruction prefetches. Many

studies have triggered sequential prefetching on a cache miss, bringing in additional cache lines after the miss target [67, 60]. A variant of this is to trigger on a miss or a *delayed hit* on the assumption that the next access will likely be a miss. Other studies have triggered such prefetching on a cache reference, counting on the additional fetch run-ahead to hide latency from misses further down the execution path [67, 60, 66]. Finally, recent studies have proposed instructions to initiate instruction prefetching [66, 45, 64].

Each of these policies represents a tradeoff between prefetch timeliness, cache pollution and miss coverage. Triggering prefetches on a miss reduces spurious cache traffic and reduces the amount of pollution that may be caused by prefetching. Triggering on a cache reference trades off these benefits to obtain better timeliness. A miss triggering policy guarantees that some cache misses will occur. A reference triggering policy attempts to hide those misses behind additional prefetching.

Software prefetch triggers can be viewed as a compromise between these positions. We would like to hide all cache miss latency but we desire low overhead in terms of cache bandwidth requirements while maintaining good miss coverage. The hope with software triggers is that the compiler can have a good idea of where cache misses may occur and therefore can schedule software triggers an appropriate distance away to achieve reasonable timeliness and coverage.

Table-based schemes such as BHGP can view this policy as a strategy for updating the prefetch table. It may be beneficial to update the table on a delayed hit as well as on a full miss because it may trigger prefetching beyond the delayed hit to targets that experience a full cache miss.

Studies such as Luk and Mowry's Cooperative Prefetching combine hardware and software triggering policies [45]. This introduces even more variation as either the hardware or

software triggers may be varied to trade off characteristics.

Replacement

For table-based hardware prefetchers such as BHGP, a related policy determines how table updates proceed. The *replacement* policy determines when and how the prefetch table is updated. Broadly, we assume as in previous work that a least-recently-used strategy gives good results. However, the question of when to update the LRU stack remains. We could update it on every access to the prefetch table, whether that access actually generates prefetches or not[2]. Another strategy only updates the LRU stack if the prefetch target is not in the cache. A third strategy may additionally update it on a delayed hit to a prefetch target.

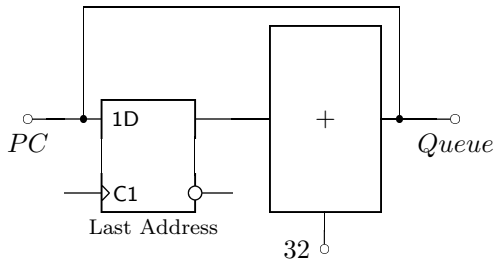
Updating on every access to the table would tend to keep around prefetch mappings that are not useful, potentially pushing out less frequently used but more desirable mappings that result in cache misses. Updating on a miss or delayed hit may produce a higher ratio of useful prefetch requests. Note that this policy is distinct from the initiation policy. It is entirely possible that a table-based prefetcher may update its prefetch table only on a full cache miss but update the LRU stack on a miss or delayed hit.

Prefetch Generation

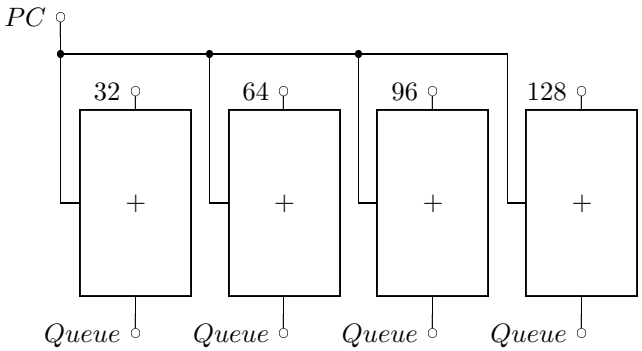
Once a prefetch has been triggered, it may cover multiple cache accesses. For example, Cooperative Prefetching assumes that the hardware sequential prefetcher will fetch eight cache lines at a time. Software prefetches in this study will fetch four cache lines. The sequential prefetcher of Xia and Torelles attempts to run far ahead of program execution.

The hardware designer must decide how many resources to devote to prefetch address

[2]Due to finding targets in the cache, for example



(a) Sequential



(b) Parallel

Figure 6.3: Request Generation

generation. If timeliness is a critical factor, the designer may choose to generate addresses for multiple requests in parallel as shown in figure 6.3(b). Alternatively, hardware may be saved by employing the sequential generation strategy of figure 6.3(a). The obvious tradeoff is the additional cycles to feed prefetch addresses to the cache.

Prefetch Sequence Termination

If a prefetch can generate multiple requests to the cache, the issue of cache pollution arises. If a sequential prefetch run-ahead strays far from actual program execution[3] the prefetcher may unnecessarily use cache bandwidth and in addition pollute the cache with useless instructions.

The designer may want to incorporate policies to terminate such prefetch sequences prematurely. Alternatively, it may be desirable to allow these sequences to continue and "drain" out of the prefetch queue to achieve some wrong-path prefetching benefit [68].

One possible termination strategy, used by Xia and Torellas, is to terminate (and redirect) prefetching whenever a branch off of the prefetch sequence is taken [66]. The goal of this strategy is to quickly redirect prefetching to the most relevant parts of the program.

Prefetch Scheduling

A multilateral prefetching scheme attempts to use various mostly-orthogonal strategies to obtain good cache miss coverage through targeted application of a variety of prefetching schemes. Such systems imply that a scheduling policy must be designed to decide which prefetcher is allows to access the cache on any given cycle. For example, the Cooperative Prefetching scheme of Luk and Mowry may generate both software- and hardware-triggered prefetches. Each potential cache access slot must be assigned to a hardware or software

[3] If, for example, a branch over a large amount of code is taken.

prefetch.

Prefetch Prioritization

The cache may have policies to determine how multiple request types are prioritized. Intuitively, it seems that always giving priority to service demand misses first would give the best performance as a demand miss represents a true program bottleneck. However, some have argued that such a policy renders prefetches useless in some cases because they wait too long in the cache MSHRs and lose any timeliness they might have had [69]. A second policy may simply handle requests in a FIFO order, not giving preference to any one type of access. Prioritizing prefetch requests is another option, however, it is possible that such a design would delay demand miss service for too long.

Prefetch Timing

Given all of the above policy variation points, it is necessary to examine the timing impacts of each selection. To do so we model an abstract prefetch mechanism using a combination of processor action blocks and cycle slots. Each action block specifies some task necessary to accomplish an instruction prefetch, such as a target address decode or a table lookup. Cycle slots are used to indicate which tasks may operate concurrently and which must occur in sequence. If two action blocks appear in the same cycle slot, they may run in parallel. time between actions is indicated by placing them in different cycle slots. For example, unit one delay between table lookup and prefetch request generation models the time necessary to access the table. The prefetch can be generated and sent to the cache in the next cycle. Arrows between action blocks indicate the flow of actions but do not imply and delays or other timing characteristics. We present these models for the

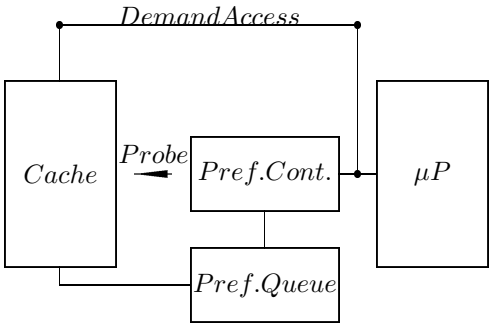


Figure 6.4: Prefetch Architecture

prefetchers studied in section 6.5.

Because so many design variation points exist, it is beyond the scope of this chapter to fully explore the potential design space. Section 6.5 outlines the areas of this space that we wish to investigate and notes potential configurations which may be interesting to explore in future work.

Architectural Model

Given the above discussion, we can present a high-level model of the general instruction prefetching architecture we will be evaluating. Figure 6.4 presents the high-level model. The core processor sends demand access to the primary instruction cache. The prefetch controller may also see those accesses depending on the initiation policy under consideration. The cache sends information to the prefetch controller about the access status (hit or miss) and the controller may probe the cache to find out if a prefetch target is already in the cache. This probe also represents the useless prefetch filtering mechanism. Once the prefetch controller determines that it should initiate a prefetch it sends the target address to the prefetch queue which is then responsible for scheduling the cache access.

```
// For each basic block in the function, place a prefetch
// distance or greater away along all paths.
for(listbb_iter b = function.blocksBegin();
    b != function.blocksEnd();
    ++b) {
  Function::instructionIterator placement(b, (*b)->instructionsBegin());
  targetType target(b);

  placePrefetchesInBlock(placement, getPrefetchDistance(), target,
                         doInterproceduralPrefetching(), IntraProcedural);
}

void InstructionPrefetch::
placePrefetchesInBlock(const Function::instructionIterator &placement,
                       unsigned int distance, targetType &target,
                       bool placeInCallers, placementCategory placementCat)
{
  listbb_iter block = target.getInstruction().getBasicBlockIterator();
  pathPush(block);
  if (!(*block)->is_Exit()) {
    followInPathFilter(getPath(), target);
    placePrefetchInBlockHelper(placement, distance, target,
                               placeInCallers, placementCat);
  }
  pathPop();
}
```

Figure 6.5: Generalized Prefetching Driver Algorithm

6.4 Software Algorithms

All of our prefetching algorithms were fully implemented in the MIRV C/C++ compiler. We chose to use the PISA architecture for these studies because it presents a generalized RISC-like instruction set and is easily modifiable. The latter feature allows us to add the necessary prefetch instructions and annotations to the machine model in a straightforward manner. We have modified the stock SimpleScalar assembler and linker to support our prefetch instructions, which essentially implement the `pf_d` prefetch instruction introduced by Luk and Mowry. The `pf_d` instruction simply encodes an address to prefetch. We provide multiple instructions to allow various PC-relative addressing modes, though in practice only the immediate addressing mode is used by the compiler.

```
unsigned int InstructionPrefetch::
placePrefetchInBlockHelper(const Function::instructionIterator &placement,
                           unsigned int dist,
                           const targetType &theTarget,
                           bool placeInCallers,
                           placementCategory placeCat)
{
  listbb_iter block = placement.getBasicBlockIterator();
  listinst_iter insn = placement.getInstructionIterator();

  // Instruction we will put the prefetch before.  Usually this will
  // be the first instruction in the block, but for interprocedural
  // prefetching it may be in the middle (see below).
  listinst_iter afterInsn(insn);
  unsigned int numInsn =
    countInstructions(afterInsn, block, dist, theTarget);

  // Find the top-most instruction in this block or the
  // first instruction after a function call that appears
  // earlier in this block (so we don't place prefetches
  // too far away).
  afterInsn = findPlacementInstruction(block, afterInsn);
  unsigned int remainingDistance = dist > numInsn ? dist - numInsn : 0;
  placementFilterResultType placementFilterResult = NOT_FILTERED;
  lastPrefetchPlacementCategory = placeCat;
  if (remainingDistance == 0
      && ((placementFilterResult = filterPlacement(
             placementType(Function::instructionIterator(block, afterInsn),
                           getCoverage()), theTarget)) == NOT_FILTERED)) {
    insertPrefetch(Function::instructionIterator(block, afterInsn),
                   getCoverage(), theTarget, placeCat);
    remainingDistance = 0;
  }
  if (remainingDistance > 0
      || placementFilterResult == FILTERED_KEEP_PREFETCHING) {
    remainingDistance = placeInPredecessors(block, remainingDistance,
                                            theTarget, placeInCallers,
                                            placeCat);
    if (doInterproceduralPrefetching() && (*block)->is_Entry()
        && placeInCallers) {
      placePrefetchesInCallers(block, remainingDistance);
    }
  }
  return(remainingDistance);
}
```

Figure 6.6: Generalized Prefetching Algorithm

```
unsigned int placeInPredecessors(listbb_iter block,
                                 unsigned int remainingDistance,
                                 const targetType &theTarget,
                                 bool placeInCallers,
                                 placementCategory placeCat)
{
  // Examine all predecessor blocks
  unsigned int newDistance = remainingDistance;
  unsigned int newRemainingDistance = 0;
  for(bb::listbbiter_iter p = (*block)->predBegin();
      p != (*block)->predEnd();
      ++p) {
    pathPush(*p);
    if (!filterPath(getPath(), theTarget)) {
        followInPathFilter(getPath(), theTarget);
        unsigned int localRemainingDistance =
      Function::instructionIterator placement(*p, (**p)->instructionsEnd()),
      placePrefetchInBlockHelper(placement, newDistance, theTarget,
                                 /*placeInCallers =*/placeInCallers,
                                 placeCat);
      // If multiple predecessors, assume we take the shortest path
      newRemainingDistance = std::max(newRemainingDistance,
                                      localRemainingDistance);
      remainingDistance = newRemainingDistance;
    }
    pathPop();
    --level;
  }
}
```

Figure 6.7: `placeInPredecessors` Algorithm

```
unsigned int placePrefetchesInCallers(listbb_iter block,
                                      unsigned int remainingDistance)
{
  // Top of function, find all of our callers and try to place there.
  insnIterList callSites;
  getCallSites(callSites, *(*block)->getHomeFunc());
  unsigned int newDistance2 = remainingDistance;
  unsigned int newRemainingDistance2 = 0;
  for(insnIterList::iterator c = callSites.begin();
      c != callSites.end();
      ++c) {
    pathPush(c->getBasicBlockIterator());
    if (!filterPath(getPath(), theTarget)) {
      followInPathFilter(getPath(), theTarget);
      unsigned int localRemainingDistance =
      placePrefetchInBlockHelper(*c, newDistance2, theTarget,
                                 /*placeInCallers = */true,
                                 InterProceduralCall);
      newRemainingDistance2 = std::max(newRemainingDistance2,
                                       localRemainingDistance);
      remainingDistance = newRemainingDistance2;
    }
    pathPop();
    --level;
  }
  return(remainingDistance);
}
```

Figure 6.8: `placePrefetchesInCallers` Algorithm

```
unsigned int countInstructions(listinst_iter &afterInsn,
                               listbb_iter block,
                               unsigned int dist,
                               const targetType &theTarget)
{
  // Walk backward through this block looking for function calls
  for(bb::reverseInstructionIterator i = bb::reverseInstructionIterator(insn);
      i != (*block)->instructionsREnd(); ++i) {
    if (isRealInstruction(**i)) {  // No pseudo-ops etc.
      ++numInsn;
      // Check for calls.  If so, try to place the prefetch there.
      if ((*i)->getPreloweredOpcode() == funcCall
          || (*i)->getPreloweredOpcode() == funcICall) {
        if (!doInterproceduralPrefetching()
            || ((*i)->getPreloweredOpcode() == funcCall
                && !(*i)->getCalledFunc()->getDefined())) {
          // For functions we can't see
          numInsn += ShortProcedureLength;
        }
        else {
          if (stepOverFunctionCalls()) { countFunction(i); }
          else {
            // If we're already at our prefetching distance, do NOT walk
            // into the callee.
            if (dist < numInsn && numInsn > 1) { afterInsn = i.base();
                                                 break; }
            bool smallProcedure = false;
            unsigned int remainingDistance =
              placePrefetchesInCallees(smallProcedure, block, i,
                                       dist - numInsn, theTarget);
            // Prefetches were all placed in callees?
            if (!smallProcedure) { return(0); }
            else { numInsn += (dist - numInsn) - remainingDistance; }
          }
        }
      }
    }
    afterInsn = --i.base();
    // Hit target distance ?
    if (numInsn >= dist) { return(numInsn); }
  }
  return(numInsn);
}
```

Figure 6.9: `countInstructions` Algorithm

```
unsigned int countFunction(bb::reverseInstructionIterator i)
{
  // Emulate the Cooperative Prefetching Algorithm
  unsigned int numInsn = 0;
  if ((*i)->getPreloweredOpcode() == funcCall) {
     unsigned int pathLength = shortestPath(*(*i)->getCalledFunc());
     numInsn = pathLength;
  }
  else {
    // Indirect call -- assume short function
    numInsn = ShortProcedureLength;
  }
}
```

Figure 6.10: `countFunction` Algorithm

We have a developed a generalized instruction prefetching algorithm that can support several instruction prefetching schemes. The generalized algorithm is listed in figure 6.6. Its driver algorithm is presented in figure 6.5 and supplemental utility algorithms are listed in figures 6.9 6.10 6.7, 6.8 and 6.11.

The generalized algorithm extends the filtering concepts of the Cooperative Prefetching algorithm (`LocalityLikely`, `HardwarePrefetched` and `SoftwarePrefetched`) by introducing an explicit filtering mechanism. Filters take one of two forms. *Path filters* prune the control-flow graph scheduling search space by eliminating paths deemed unimportant by various heuristics. An example is the `sameBlock` filter which prevents the walk from traversing into a block multiple times for the same target (c.f. line `A` of figure 6.2). *Placement filters* prevent the scheduling of prefetches in undesirable locations. Various filters implement equivalents to `LocalityLikely` and parts of `SoftwarePrefetched`.

`HardwarePrefetched` and parts of `SoftwarePrefetched` are special cases. Because of their placement in the Cooperative Prefetching algorithm (c.f. line `B` of figure 6.2) they halt traversal through the control-flow graph. Thus they are implemented as path filters in our algorithm. Bits of `SoftwarePrefetched` are also implemented as placement filters (to

```
unsigned int placePrefetchesInCallees(bool &smallProcedure,
                                      listbb_iter block,
                                      listinst_iter &i,
                                      unsigned int dist,
                                      const targetType &theTarget)
{
  insnIterList returnSites;
  getReturnSites(returnSites,
                 Function::instructionIterator(block, --i.base()));
  unsigned int remainingDistance = 0;
  for(insnIterList::iterator r = returnSites.begin();
      r != returnSites.end();
      ++r) {
    pathPush((*r).getBasicBlockIterator());

    if (!filterPath(getPath(), theTarget)) {
      followInPathFilter(getPath(), theTarget);
      unsigned int localRemainingDistance =
        placePrefetchInBlockHelper(*r, dist, theTarget,
                                   // We'll clean up if too short
                                   /*placeInCallers =*/false,
                                   returnSites.size() > 1 ?
                                   InterProceduralMultiReturn :
                                   InterProceduralReturn);

      if (localRemainingDistance != 0) {
        // Function too small, continue counting instructions
        // If there are multiple return sites, assume we took
        // the shortest one.
        remainingDistance = std::max(remainingDistance,
                                     localRemainingDistance);
        smallProcedure = true;
      }
    }
    pathPop();
    --level;
  }
  return(remainingDistance);
}
```

Figure 6.11: `placePrefetchesInCallees` Algorithm

check for duplicate prefetches in a block, for example).

This filtering framework has also proven flexible enough to implement many of the prefetch optimizations described by Luk and Mowry. We have implemented their combine-at-dominators optimization. We have also implemented a subset of their unnecessary prefetch optimization, which attempts to remove prefetches for blocks already in the cache. This subset does not cover the cases of cycles introduced by mutually recursive procedures.

6.4.1 A Software Instruction Prefetching Framework

Given the generalized algorithm above, we can begin to think about how concrete prefetching algorithms are expressed through the use of the generic filtering mechanism. We noted above that the various software filters used by Luk and Mowry can be mapped into placement and path filters in the general algorithm. Similarly, the Software CGP algorithm can be broken down into a scheduling algorithms that utilizes filters to prune the search space. The Software CGP filters are relatively simple. No particular path is pruned, but prefetches may only be placed at very specific locations: either immediately after a procedure call or at one of the equidistant points throughout the procedure.

Given these mappings, we have developed a framework to concisely express the characteristics of software instruction prefetching algorithms. All such algorithms possess some sort of scheduling phase which attempts to place prefetches such that they are useful and timely. The main point of variation is the heuristics to determine these two qualities for a particular placement point. In addition, the path filtering mechanism can reduce or eliminate unnecessary searches.

In our framework, software instruction prefetching algorithms are expressed as a five-tuple. Each element represents a variation point among prefetching algorithms. The first

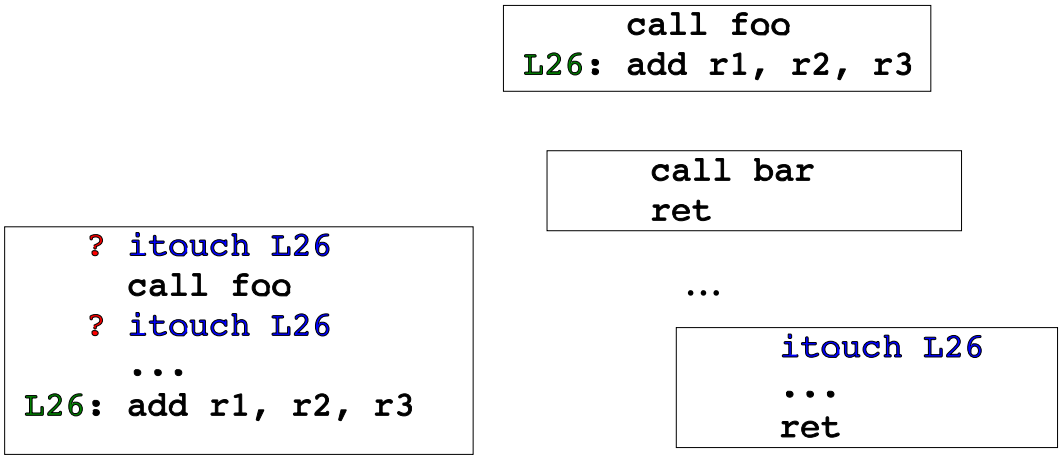
tuple element describes the target the algorithm attempts to prefetch. Ultimately, of course, prefetches target cache lines. However, each algorithm may represent this idea in a different way. For example, the Cooperative Prefetching algorithm targets basic blocks in the program. Each prefetch instruction placed is assumed to reference the beginning of some basic block. Many of the filtering mechanisms depend on recognizing which basic blocks prefetches target. The Software CGP algorithm targets both procedures and cache lines. Each prefetch placement after a procedure call targets some other procedure. The equidistant prefetches implicitly target cache lines.

The second tuple element in our framework describes the scope of the scheduling algorithm. By *scope* we mean the range over which instructions may be scheduled. We identify two distinct scopes: an intra-procedural scope and an inter-procedural scope. Both Cooperative Prefetching and Software CGP are intra-procedure scheduling algorithms because the scheduling algorithm does not walk backward into other functions when it encounters call instructions or if it reaches the top of the current procedure.

Note that the scope of the algorithm has nothing to do with the scope of the prefetch targets. Both Cooperative Prefetching and Software CGP can target items in procedures outside of the procedure being scheduled. It is the scheduling algorithm itself that is described by this tuple parameter.

The third item is related to the scope. We have dubbed this the *step* characteristic. In an inter-procedural algorithm, a decision must be made about how to schedule at procedure call points. If the scheduling algorithm reaching a procedure call it has the option of stepping into the called procedure[4] and continuing the control-flow graph walk or it may step over the call and continue scheduling within the current procedure. We have called

[4]This may imply multiple procedures for an indirect call.



(a) Step Over

(b) Step Into

Figure 6.12: Prefetch Algorithm Step Policy

this the *step* parameter due to the analogy with stepping into or over a call in an interactive debugger. Figure 6.12 illustrates the two strategies for inter-procedural schedulers. Figure 6.4.1 describes a "step over" policy in which prefetch instructions are scheduled before or after a procedure call but are not scheduled within the callee function. Figure 6.4.1 shows a "step into" policy which can schedule prefetches inside the body of the caller or, as in this figure, inside a deeply nested procedure invoked during execution of the callee.

The fourth item of the tuple is a Boolean expression describing the placement filter used by the algorithm. This is by necessity an abstract description as providing details about the filter algorithms would complicate the description. As an example, the Cooperative Prefetching placement filter can be described by the expression

$$\neg sufficient\ distance \vee prefetch\ exists \vee locality \vee dominator$$

The above expression assumes known definitions for the terms used. In this case *sufficient distance* refers to the run-time distance in instructions between the placement point

and target point. The *prefetch exists* item indicates whether another prefetch with the same target already exists at the placement point. The *locality* parameter specifies the `LocalityLikely` algorithm of Luk and Mowry, which attempts to determine whether the placement and target points can coexist in the cache. Similarly, the *dominator* item refers to the various dominator optimizations performed by Cooperative Prefetching to reduce the number of redundant prefetches by moving them into dominator blocks.

The final tuple item specifies the path filter used by a particular algorithm. It is also a Boolean expression. The path filter specifier for Cooperative Prefetching is

$$sequential\ prefetch \vee interprocedural$$

This expression describes the pruning of prefetch scheduling paths if a parent block is determined to be covered by the hardware sequential prefetcher or if the block exists in another procedure. Recall the Cooperative Prefetching is an intra-procedural algorithm in terms of scheduling instructions. In this case we only allow scheduling of an inter-procedural prefetch for the entry block of the target function and we only allow scheduling through one level of caller. Another way to view this is that the algorithm examines all functions called in a procedure and schedules prefetches for those functions within the body of the current (caller) procedure. Thus the scheduling is intra-procedural.

6.5 Methodology

In this section we describe our experimental methodology. We begin by presenting our implementations of the various prefetching architectural variation points discussed in sections 6.3. We then describe our simulation environment and list the machine models used to evaluate the utility of the prefetching schemes. All of our software prefetching simulations

use the generalized algorithm presented in section 6.4, tuned with filters appropriate to each specific prefetching strategy. These filters are also discussed in this section.

6.5.1 Prefetch Architecture

In section 6.3 we described several design variation points for hardware prefetching architecture. Because of the large design space implied by these variation points, we have necessarily chosen a subset of possible designs for our experiments. The primary motivation is to determine the performance impact of these variations.

Cache Interface

As mentioned in section 6.3.2 we study two prefetch queue designs: the straightforward FIFO (Default) and the sequential prefetching queue proposed by Xia and Torellas (Advanced). We model these as infinite queues because discussion with Todd Mowry indicated the importance of not dropping any prefetch requests [70]. In addition, we include two filters to reduce the number of useless prefetches sent to the cache. We employ the obvious policy of querying the cache tags before issuing a prefetch, known as Cache Probe Filtering [71]. This query uses a cache port during the cycle in which it occurs, meaning there is one less port available for prefetching or demand accesses from the fetch engine. Our other filter is identical to that proposed by Luk and Mowry in their Cooperative Prefetching work. This filter maintains a two-bit saturating counter for each secondary cache line. If a prefetch brings such a line into the primary cache and that line is subsequently evicted without having been accessed, the corresponding counter is incremented. An access to the prefetched line resets the counter. If the counter rises above some threshold (two in our studies) prefetches to the line are squashed.

Our model keeps track of the number of instruction cache ports used by the fetch engine and will only issue prefetches to the cache if a port is available. In addition, we explore the design where an unlimited number of cache ports is available to check for in-cache prefetches. The motivation is two-fold. The studies of Luk and Mowry indicate that their prefetch engine performs such cache probes but it is not clear whether such probes were accounted cache ports. We also wish to determine whether such prefetches unnecessarily delay cache access for useful prefetches by clogging up the prefetch request queue.

In addition to these elements, all of our experiments assume a 16-entry prefetch buffer [72]. Prefetches insert cache lines into this buffer and demand accesses move it into the primary cache. When the buffer becomes full, the oldest entries are moved into the primary cache as outlined by Luk and Mowry [73]. The prefetch buffer acts as an element in a multilateral cache composed of the primary cache, prefetch buffer and an eight-entry victim cache [72]. It is important to note that this architecture is quite different from that studied in the BHGP paper. The architecture there has a 16K primary instruction cache and a 2K, 4-way associative prefetch buffer that likely does not operate as a queue, though that is not clear from the published work. It is possible that cache pollution may be a greater problem for BHGP in our experiments as compared to the previously published work. However, we wish to maintain a consistent environment so that we may most fairly evaluate all of the prefetching schemes.

Prefetch Initiation

Our experiments explore the two main prefetch initiation policies of section 6.3.2: prefetch-on-miss (FullMiss), prefetch-on-delayed-hit (Miss) and prefetch-on-reference (Reference). Software prefetch triggers are handled immediately after instruction decode and

prefetch instructions do not progress further down the pipeline so that they do not use hardware resources unnecessarily.

For the BHGP prefetching scheme, this policy controls when branch addresses may be associated with cache misses. Only the FullMiss and Miss policies are relevant. The policy controls when the M bit in the prefetcher is set.

Replacement

For the table-based schemes of BHGP, CHGP and Cooperative CHGP, we study three replacement strategies. The first, Hit, updates the LRU stack on every table access, whether or not that access actually generates any prefetches. The second, Miss, updates on a delayed hit or a full cache miss. The third, FullMiss, only updates on a full cache miss. For prefetch sequences greater than one cache line, any prefetch target line that experiences a delayed hit or a full miss will cause an LRU update with these policies.

Prefetch Generation

We explore designs that generation multiple-prefetch addresses in sequence (Sequential) and in parallel (Parallel). Generating such addresses in parallel allows prefetches to access the cache sooner. In particular, a specific prefetch may immediately use multiple available cache ports if the addresses can be generated quickly enough.

Prefetch Sequence Termination

For designs that use the advanced sequential prefetch design of Xia and Torellas, we explore two policies to terminate and redirect prefetch sequences. The first policy, *Drain*, simply allows the prefetches generated by a sequence to remain in the request buffer and drain out as they perform their cache access. Sequence generation is redirected on a cache

miss or when software prefetch instructions are encountered.

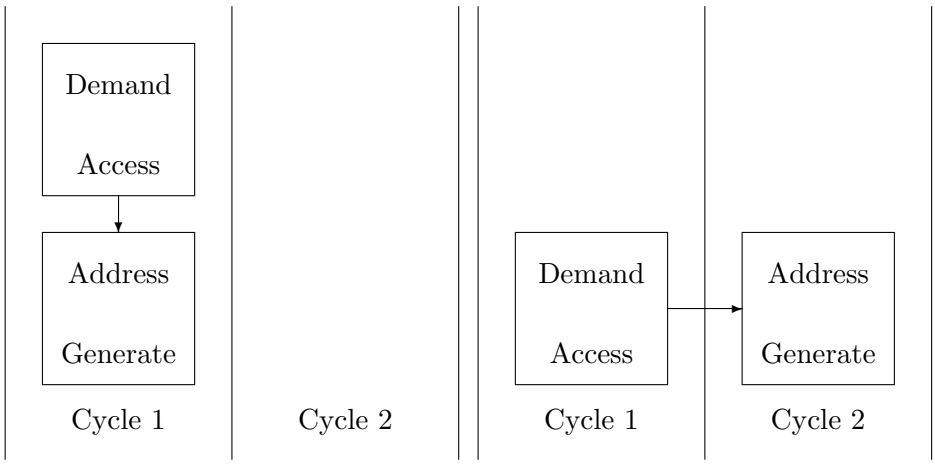
Our second policy, *Branch*, attempts to quickly redirect prefetch generation when branching is encountered. Any prefetches being generated by the current sequence are squashed when a branch to a location outside the current sequence (delimited by the starting and ending addresses of a sequential prefetch request) is encountered. The prefetch queue is emptied so that the sequence starting with the branch target can immediately begin accessing the cache. In addition, outstanding requests for prefetch address generation (under the *Sequential* generation policy) are squashed.

Note that the advanced prefetch queue always redirects prefetch generation on a cache miss or a branch. This policy simply states what happens to existing prefetch requests in the request buffer. Software prefetches do not redirect the sequential prefetch engine.

Prefetch Scheduling

We examine the use of two prefetch scheduling algorithms. Scheduling only comes into play when hardware and software prefetchers are present. Therefore, experiments which only explore hardware prefetching are not affected by this policy.

The first policy, *FIFO*, sends prefetch requests to the cache on a first-come, first-served basis. The *RoundRobin* policy maintains separate prefetch request queues for software and hardware prefetches. Scheduling ping-pongs between the two queues. If one queue becomes empty its scheduling slots may be used by the other queue in that cycle. This policy places additional importance on software prefetches under the assumption that the compiler has placed such a prefetch because it is important to issue to the cache as soon as possible. However, we do not wish software prefetches to starve the hardware prefetcher. The round-robin policy attempts to balance these concerns.



(a) Reference (b) Miss

Figure 6.13: Prefetch Initiation Architectures

Prefetch Prioritization

In this work we do not explore the trade-offs involved in prioritizing demand misses over prefetches or vice-versa. Our cache processes all accesses in a first-come, first-served fashion. There is some disagreement in the research community about which policy is most effective and further study is warranted [74].

Prefetch Timing Models

In section 6.3 we described an abstract model of instruction prefetching hardware. We now present the timing models for the above policies that we wish to explore.

Depending on how prefetches are initiated we may have to delay prefetch generation from the time the instruction cache is accessed. Figure 6.13 presents a diagram of the options. With a Reference policy, prefetch address generation may proceed in parallel with instruction fetch because the machine does not care about the hit/miss outcome of the cache access. In fact the sequential prefetches may be sent to the cache at the same time as

the fetch (assuming enough ports are available) because the sequential address generation can occur at the same time as the next program counter address calculation.

A Miss policy requires that the processor wait a cycle to see whether the demand fetch hit or missed in the cache. Prefetch address generation may still occur in parallel with the fetch but the actual cache access must wait until the next cycle.

Prefetch target address generation can proceed in a Sequential or Parallel manner. The resulting architecture models are obvious. These are presented in figure 6.14. The diagram starts at cycle zero to indicate that prefetch address generation can occur in parallel with the demand access address generation. Thus with a Reference initiation policy the prefetch access may occur in parallel with the demand access. We assume that there is enough machine state to handle all sequential prefetch address generation without dropping requests.

Once a prefetch request is generated, we assume that it may proceed directly to the cache if the request buffer has fewer entries than the number of free cache ports in the current cycle. Otherwise the request is enqueued and prefetches are sent to the cache according to the scheduling policy (RoundRobin or FIFO) in effect. We do not model any additional delay for performing the scheduling.

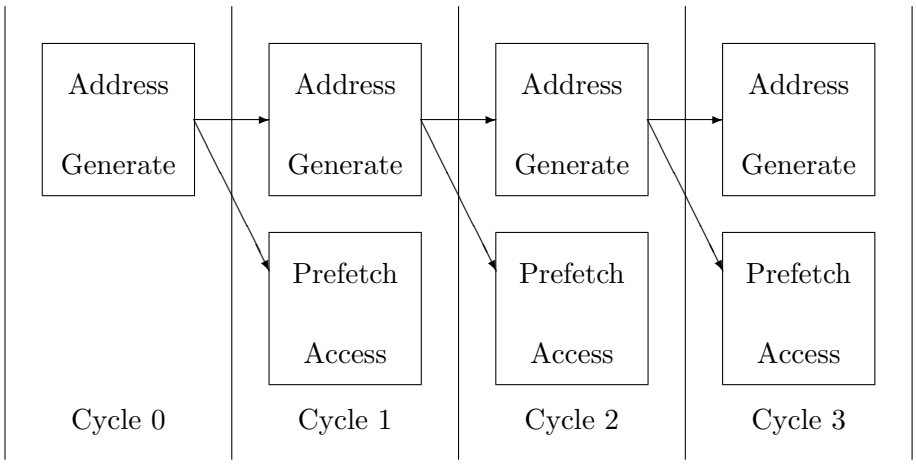
6.5.2 Software Prefetch Filters

We use a variety of different filter expressions for our experiments. The Cooperative Prefetching experiments use the filters described in section 6.4.

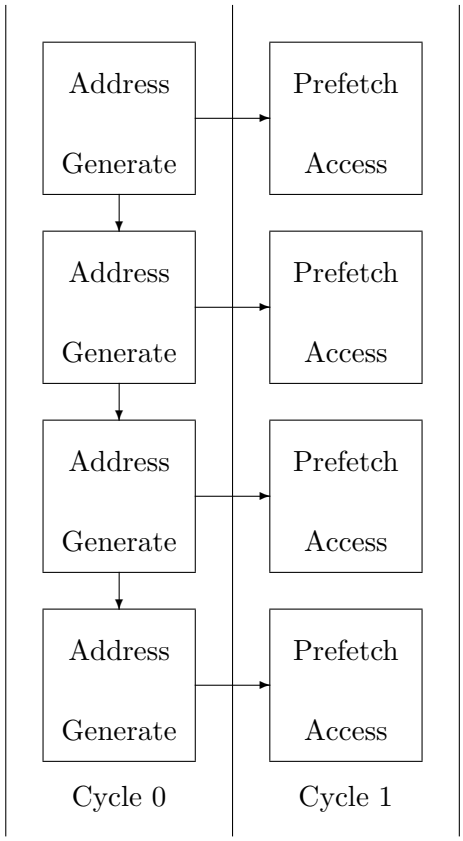
$$Placement_{CP} = \neg sufficient\ distance \vee prefetch\ exists \vee locality \vee dominator \quad (6.1)$$

$$Path_{CP} = sequential\ prefetch \vee interprocedural \quad (6.2)$$

The other algorithms use some variant of these equations. The CHGP filters are de-



(a) Sequential



(b) Parallel

Figure 6.14: Prefetch Generation Timing

scribed by the following equations:

$$Placement_{CHGP} = \neg sufficient\ distance \vee prefetch\ exists \vee locality \tag{6.3}$$

$$Path_{CHGP} = interprocedural \tag{6.4}$$

We remove the dominator filter because we want to initiate prefetching from a variety of places in the control flow graph. The dominator filter exists to reduce the bloat caused by instruction prefetch instructions. Since CHGP does not have this problem, the dominator optimization simply tends to combine multiple prefetch launch points into one and may reduce the effectiveness of the annotations. In other words, more launch points may result in better timeliness.

Cooperative CHGP uses the following equations:

$$Placement_{CCHGP} = \neg sufficient\ distance \vee prefetch\ exists \vee locality \tag{6.5}$$

$$Path_{CCHGP} = sequential\ prefetch \vee interprocedural \tag{6.6}$$

Because a hardware sequential prefetch covers the short-distance prefetches we should not mark any instructions within the sequential prefetching window. We use the path filter method employed by Luk and Mowry to prune the schedule search space.

Unlike the study performed by Luk and Mowry, our cache heuristics do not artificially shrink the instruction cache seen by the compiler. It has been our experience that the filters overestimate the size of instruction sequences anyway and so shrinking the cache will simply exacerbate those problems.

Param	Value					
Issue	out-of-order					
Width	1, 2, 4 or 8					
Fetch Buffer	16 or 512 Instructions					
IQ	32 or 1024 Entries					
LSQ	16 or 512 Entries					
Store Buffer	32 or 1024 Entries					
ROB	32 or 2048 Entries					
Branch Predictor	McFarlan Hybrid 2K 11bit local history 13bit global history 4-way 4K BTB 16 entry RAS 3 cycle mispredict penalty					
Seq. Prefetch	0, 1 or 8 lines					
Function Units	**Integer**		**Floating Point**		**Memory**	
	ALU	2	ALU	2	DPorts	2
	Mult/Div	1	Mult/Div	1	IPorts	4
Cache	**L1 Instruction**		**L1 Data**		**L2 Unified**	
	Size	32K/inf	Size	32K	Size	1M
	Assoc	2-way	Assoc	2-way	Assoc	2-way
	Line Size	32-byte	Line Size	32-byte	Line Size	32-byte
	MSHRs	32	MSHRs	32	MSHRs	32
	MSHR Tgts	16	MSHR Tgts	16	MSHR Tgts	16

Table 6.1: Simulation Parameters

6.5.3 Simulation Environment

We used the `M5` simulator in all our experiments to model various current and "near future" architectures [3]. `M5` is an event-driven simulator that fully models machine pipelines and includes a sophisticated memory model that is able to track bus contention. We modified `M5` to implement our prefetch instructions and record various metrics during execution time to evaluate the effectiveness of our algorithms.

The simulator parameters we used in our experiments are listed in table 6.1. We list multiple values for variant parameters.

6.5.4 Experiment Design

We run a series of experiments to quantify the impact of policy and architectural variations. For our simulations, the issue width can never exceed the fetch width. We simulate

two different instruction window configurations, *smallwin* and *bigwin* designs. The former uses an extremely small window extrapolated from the Cooperative Prefetching studies of Luk and Mowry. Their study specifies a "Fetch and Decode Width" of eight instructions. We supply a 16-entry fetch queue to support this. They also specify a reorder buffer size of 32 entries, which we interpret as an instruction window of 32 entries and a 16-entry load/store queue. The 64-entry ROB is increased from the specified size of 32 to accommodate the additional LSQ space. Therefore, this small window configuration is actually a bit larger than that used by Luk and Mowry. We also include a 32-entry store buffer which is not specified in the Cooperative Prefetching study. Like Luk and Mowry, all of our experiments include a prefetch buffer and victim cache which operate in parallel with the primary level-one instruction cache.

Our large window design is an attempt to remove fetch delays caused by the instruction queue filling up. This quite unrealistic design includes 512-instruction fetch buffer, a 1024-entry instruction window, a 512-entry LSQ, a 2048 entry ROB and a 512-entry store buffer. This design is studied to determine the impact of instruction prefetching in the absence of front-end structural hazards.

We run three main sets of experiments. The first is a set of baseline experiments to determine the finite instruction cache penalty experienced by the SPEC 95 and 2000 integer benchmark suites. We include studies of the big- and small-window machines to determine which machine configuration is best suited to study instruction prefetching.

Because many of the SPEC integer benchmarks are quite small and do not experience much of a finite instruction cache penalty at all, we concentrate our further experiments on the subset of benchmarks that show some potential for improvement. Given the architectural parameters determined in the previous set of experiments, we examine the impact of

the various prefetching policies on the performance of our prefetching schemes.

Finally, given the results of these policy experiments, we explore how prefetching performs on a variety of machine configurations with respect to issue width, primary-to-secondary cache bandwidth and cache ports available for in-cache prefetch checks.

6.6 Results

We ran the experiments described in section 6.5 and present the results here. We organize the results into three major categories. The first holds the results for sequential prefetching. We wish to use a sequential prefetcher as the baseline to which the other prefetch schemes are compared. We want to select the best possible sequential prefetcher so we examine a number of policy experiment results to determine a reasonable architecture.

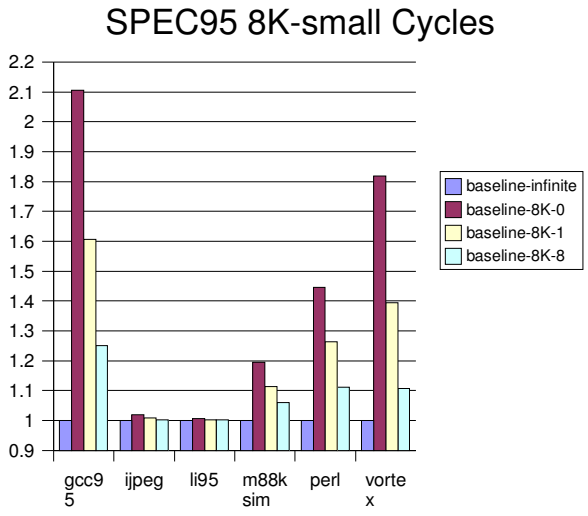
6.6.1 Baseline Results

Our baseline results compare sequential prefetching against machines with no prefetching. Ultimately the best we could ever hope to do is approach an infinite level-one instruction cache so we compare our initial runs to that configuration. We then compare various prefetching policies to determine the most effective sequential prefetching method.

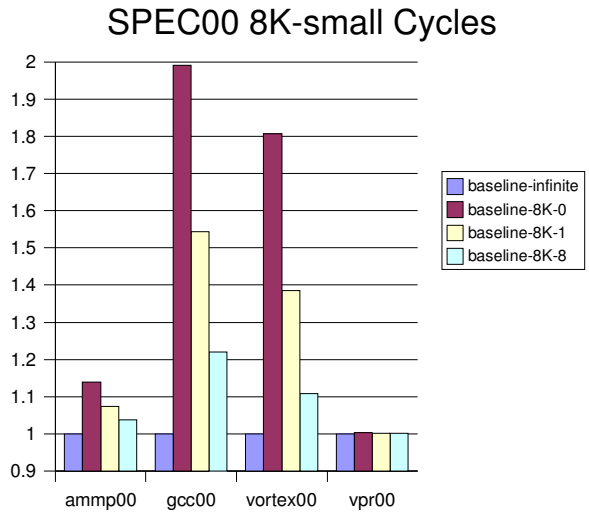
Finite Cache Penalty

Figures 6.15, 6.16, 6.17 and 6.18 show sets of data for sequential prefetching with an 8K and a 32K cache. Figures 6.15 and 6.17 show results for the small instruction window architecture while figures 6.16 and 6.18 show results for the large window architecture.

Each figure shows the relative slowdown of a particular configuration to that of an infinite level-one instruction cache. There are four bars for each benchmark. The first is the

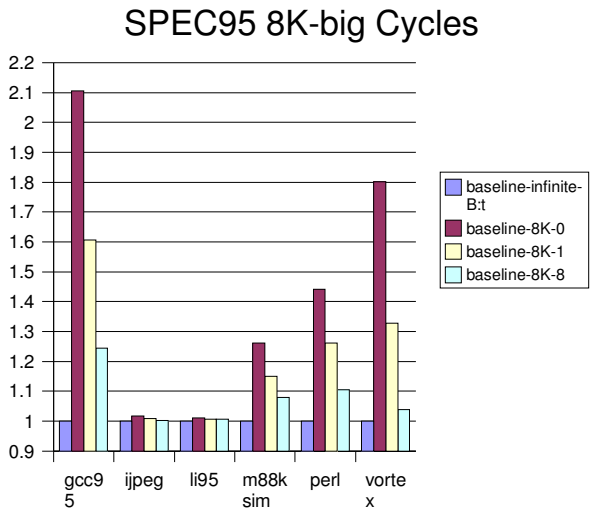


(a) SPEC 95

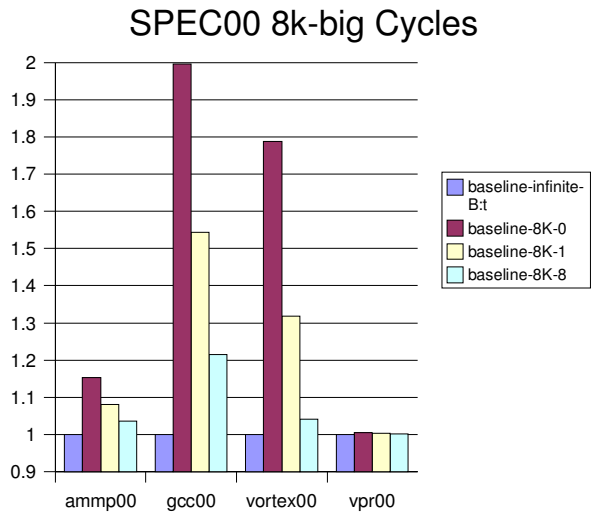


(b) SPEC 00

Figure 6.15: Baseline 8K Sequential Prefetching, Small Window

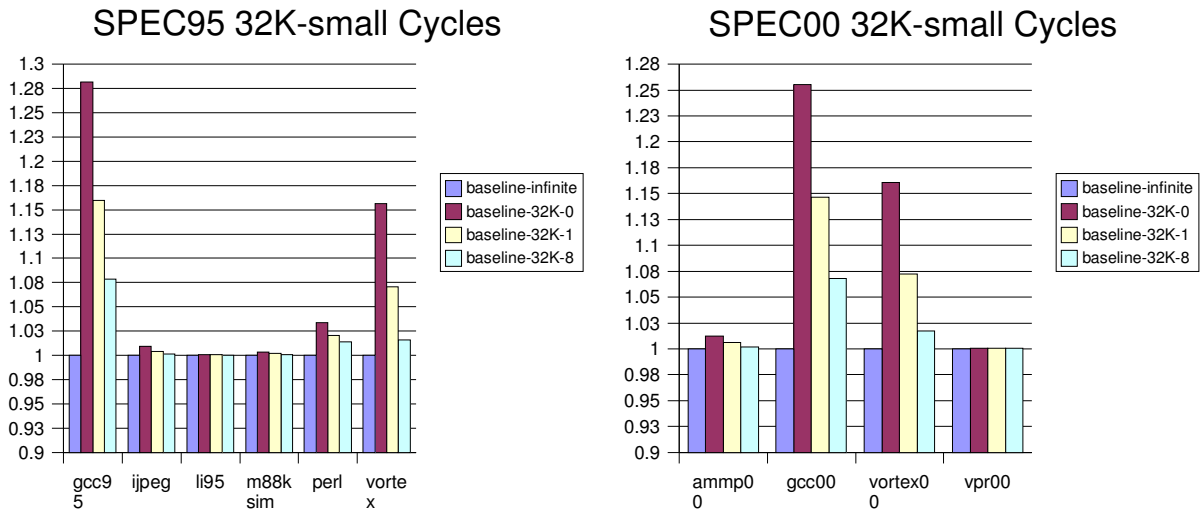


(a) SPEC 95



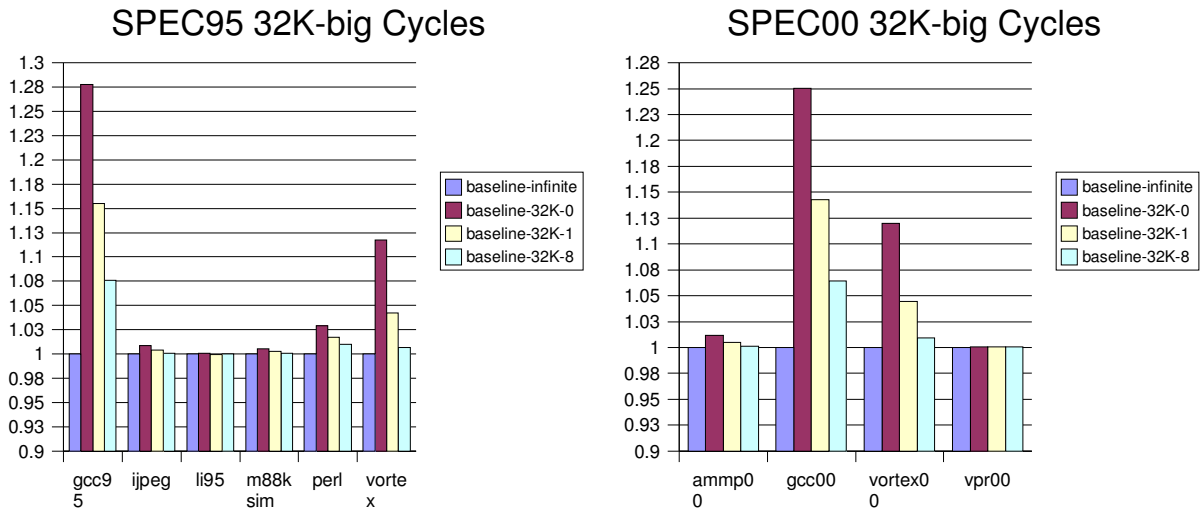
(b) SPEC 00

Figure 6.16: Baseline 8K Sequential Prefetching, Large Window



(a) SPEC 95 (b) SPEC 00

Figure 6.17: Baseline 32K Sequential Prefetching, Small Window



(a) SPEC 95 (b) SPEC 00

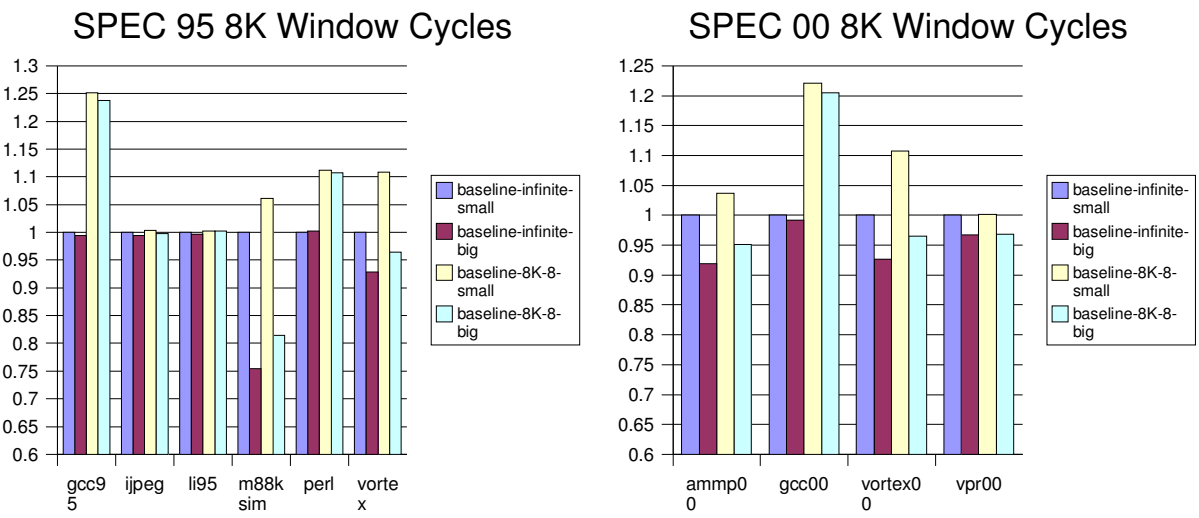
Figure 6.18: Baseline 32K Sequential Prefetching, Large Window

relative slowdown of the infinite cache configuration. It is always one. The second bar shows the finite cache penalty. The performance shown is that of a machine with the specified cache size and no instruction prefetching. The last two bars indicate the performance improvement achieved with sequential prefetching, first with one line of lookahead and then with eight lines.

All baseline experiments used the Default prefetch queue with prefetch Miss initiation and Parallel address generation. The machine can fetch up to eight instructions per cycle and issue up to four to the function units each cycle.

Most benchmarks do see a finite instruction cache penalty. Furthermore, the increased penalty for the eight kilobyte cache indicates that it is not simply due to compulsory misses. However, sequential prefetching recovers almost all of the penalty in every benchmark except perl, gcc and vortex. The variations between the SPEC 95 and SPEC 2000 flavors of these benchmarks is not terribly significant. The remainder of our experiments will focus on the perl and gcc benchmarks from SPEC 95 and the vortex benchmark from SPEC 2000 because they present opportunity for transition prefetch strategies such as BHGP and Cooperative Prefetching to improve performance. These are also the SPEC benchmarks studied by Luk and Mowry in their Cooperative Prefetching work, providing a point of comparison.

A very interesting result becomes apparent if we compare the large and small instruction window configurations. While the absolute number of cycles to execute the programs is reduced in the large window case, the relative benefit of instruction prefetching remains about the same. The vortex benchmark is improved a little more in the large window configuration while m88ksim sees slightly less improvement over prefetching with a small instruction window. There is a complex set of interactions that can affect the benefit of prefetching. On one hand, a small instruction window will be more likely to fill up, stalling



(a) SPEC 95

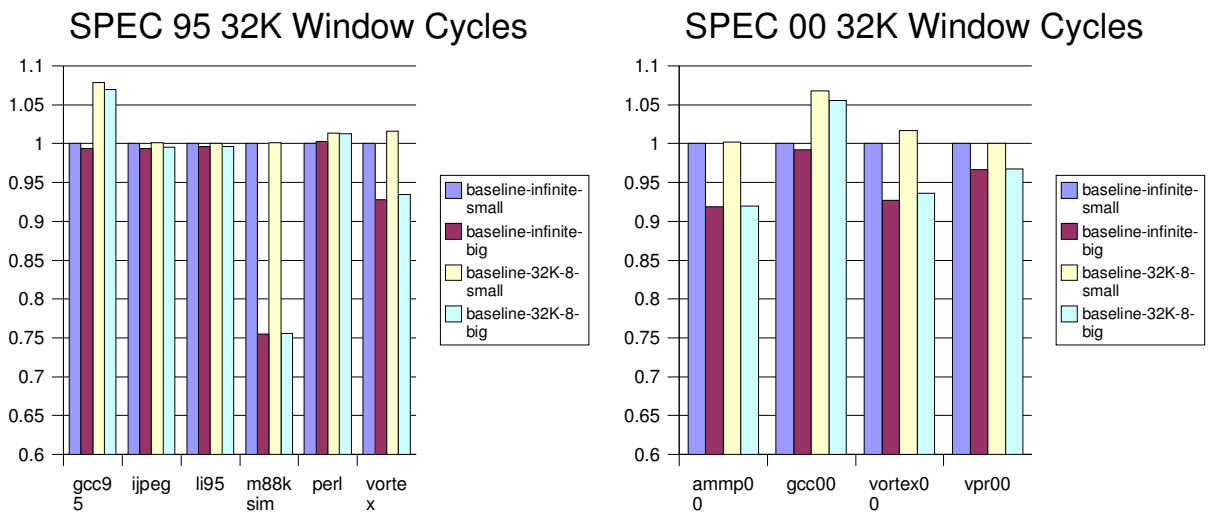
(b) SPEC 00

Figure 6.19: Baseline 8K Window Performance

the fetch stage even in the presence of cache hits. This means that any in-flight prefetches will have more time to be serviced, potentially improving prefetch timeliness. On the other hand, a full window may hide some of the prefetching benefit because the machine may experience a full-window stall the next cycle even in the presence of a cache hit in the current cycle. In other words, even if a prefetch successfully targets a cache miss, the machine may still stall.

The branch predictor can also attenuate the benefit of a large instruction window [75]. A large window does not gain much if misspeculation constantly fills it with useless instructions. Such misspeculation can also reduce the effectiveness of instruction prefetching because prefetches will be launched for incorrect targets, though such prefetches may be used if the misspeculated path is traversed on some later iteration.

If we examine the performance of the large- and small-window configurations with no instruction prefetching, we find that the large window is not giving as much improvement



(a) SPEC 95

(b) SPEC 00

Figure 6.20: Baseline 32K Window Performance

as one might expect. Figures 6.19 and 6.20 show the performance of the two window configurations given an 8K and a 32K cache, respectively. Both graphs also show the performance with an infinite cache, to isolate the effects of the window size. On gcc95, the improvement is negligible for all cache configurations. For vortex00, the cycle count is reduced by 7% in the infinite cache case, 5% in the 8K cache case and 6% in the 32K cache case. It is interesting to note that prefetching on a large window with a finite cache outperforms the infinite cache, small window machine for some benchmarks. vortex00 gets a 3% performance improvement with prefetching on a large window machine versus prefetching on a small window machine in the 8K cache configuration. The improvement is only 1% for the 32K cache configuration. The large window does not significantly perturb results for prefetching given a specific architecture, justifying our choice of architecture for the remaining experiments. If anything, the large window machine will make it more difficult for prefetching schemes to outperform sequential prefetching.

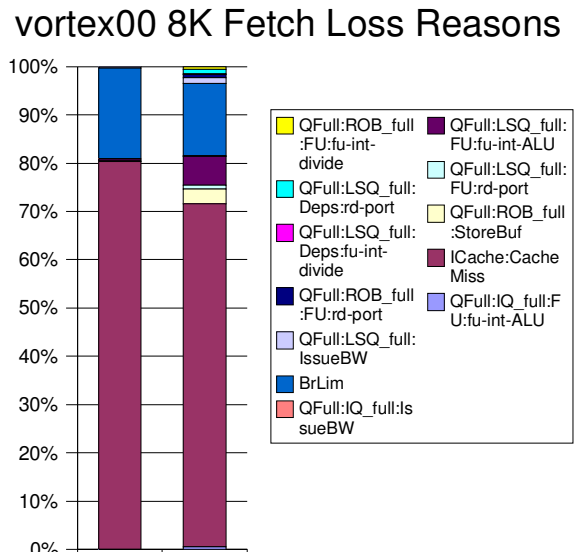


Figure 6.21: Fetch Loss Reasons

To get a handle on the impact of the small instruction window, we examined the reasons for stalling in the fetch stage. Each cycle the simulator records the reason for any fetch stall that is seen. Figure 6.21 presents this data for the SPEC 2000 vortex benchmark with no instruction prefetching and an 8K instruction cache. Column 1 shows the data for the large instruction window while column 2 shows the data for the small instruction window. The large window has removed most of the full-window stalls, leaving branch fetch limits and instruction cache misses as the dominant culprits. Because the relative improvement of sequential prefetching is not significantly affected by the choice of window size, we run further experiments assuming a large window so that we can eliminate the full-window stall component. Because the machine is not stalling due to a full instruction window, prefetch timeliness should be even more critical.

Filtering Impact

The Cooperative Prefetching work by Luk and Mowry reports that sequential prefetching is much less effective than we have shown above. They report a 5% improvement for eight-line prefetching on gcc95. Our results above indicate about a 20% improvement over the 32K cache, small window baseline performance. There are several reasons these results may differ:

- Luk and Mowry report only a 17% finite cache performance penalty while we observe a 29% penalty with a 32K cache.
- Unspecified architectural parameters (as noted in section 6.5 may differ widely between the two experiments.
- Benchmark datasets differ.
- We assume four full cache ports while Luk and Mowry use a four-banked cache.
- Our sequential prefetcher uses the same hardware filter that is used by Cooperative Prefetching. It is not clear that Luk and Mowry's baseline results employ such a filter.

The first point is closely related to the second and the second point is for all practical purposes impossible to eliminate. Without precise and accurate descriptions of the previous experiment, we cannot hope to reproduce it exactly. As for the third point, Luk and Mowry, like us, use reduced SPEC datasets. Unfortunately, we could not obtain those datasets for comparison to ours. The fourth point requires a more sophisticated memory model than is currently available in M5. Our numbers will certainly be more optimistic due to this. However, we present results below that indicate ports are not necessarily a bottleneck for distant sequential prefetching given the 32-byte data bus to the secondary cache.

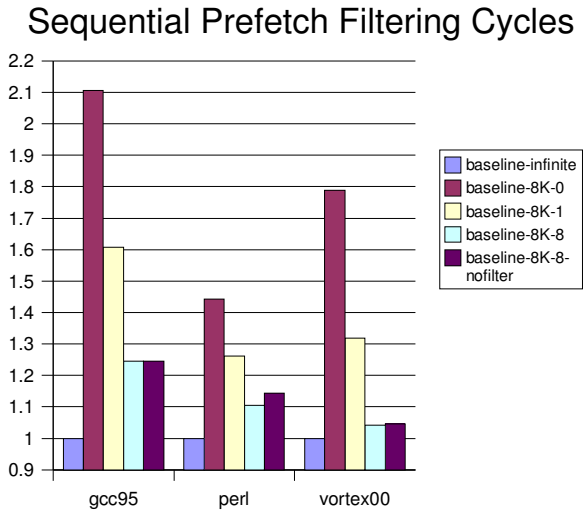
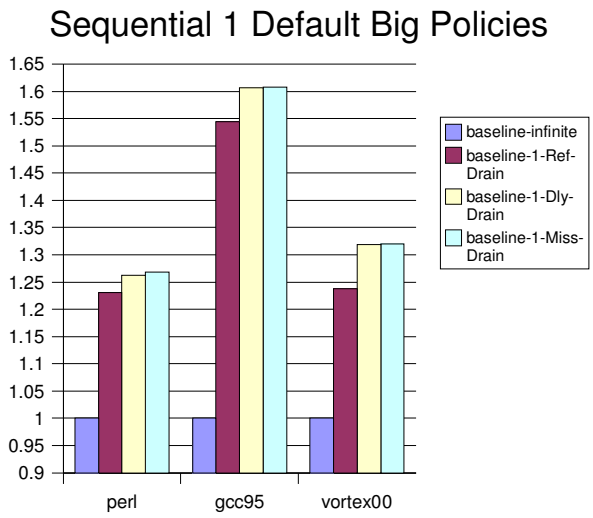


Figure 6.22: Prefetch Filter Impact

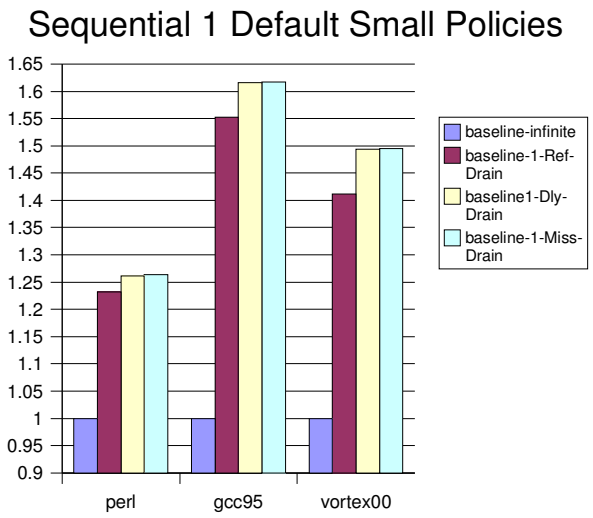
We can quantify the impact of the last point by running experiments without the hardware filtering. We perform runs of gcc95, perl and vortex00 with eight-line sequential prefetching and a large instruction window with the hardware prefetch filter disabled. Figure 6.22 shows the results. The first three columns reproduce the sequential prefetching results above. The fourth column shows the performance of sequential prefetching without hardware filtering. As expected, performance degrades in each case. However, the impact is not as severe as one might expect, leading to the conclusion that other architectural models are the primary reason for the differing results.

Policies

Figures 6.23 and 6.24 present the performance impact of the various prefetching policies on a one-line fetch-ahead sequential prefetcher. Figure 6.23 presents data for the Default prefetch request queue while figure 6.24 presents data for the Advanced queue. Both figures show performance on the big and small window machines. Similarly, figures 6.25 and 6.26 show the performance with an eight-line fetch-ahead prefetcher.

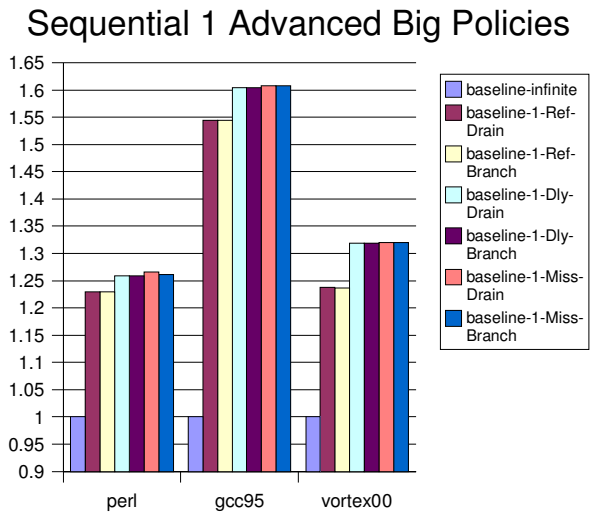


(a) Big Window

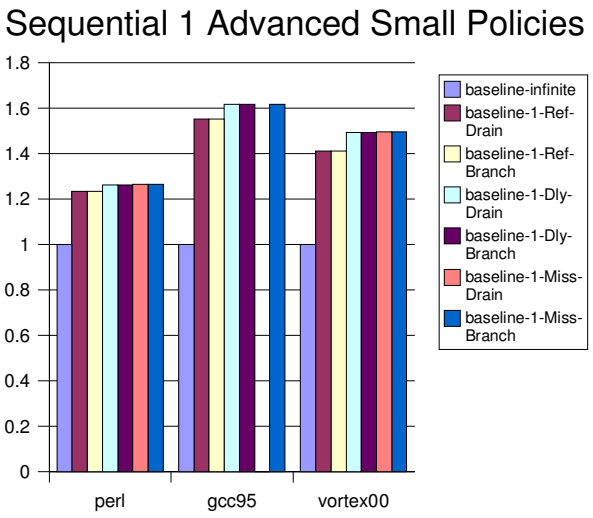


(b) Small Window

Figure 6.23: Baseline Sequential-1 Default Queue Policies Performance

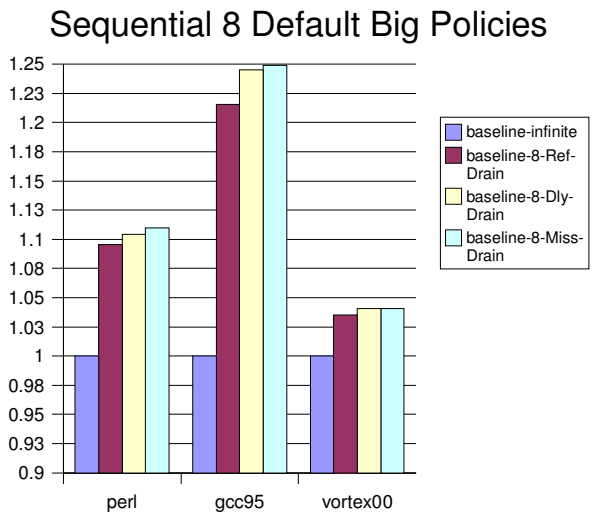


(a) Big Window

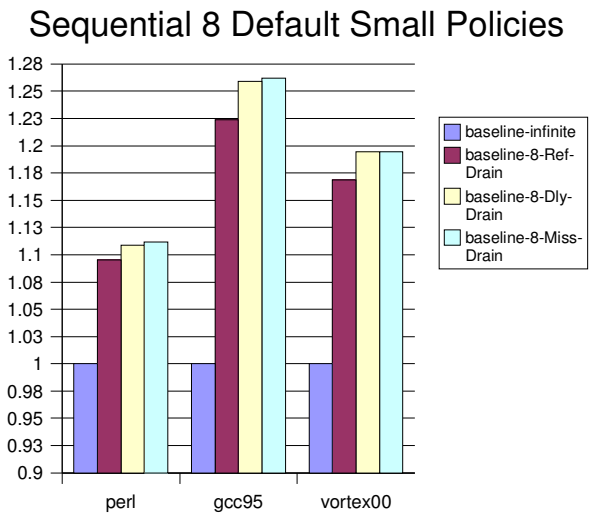


(b) Small Window

Figure 6.24: Baseline Sequential-1 Advanced Queue Policies Performance

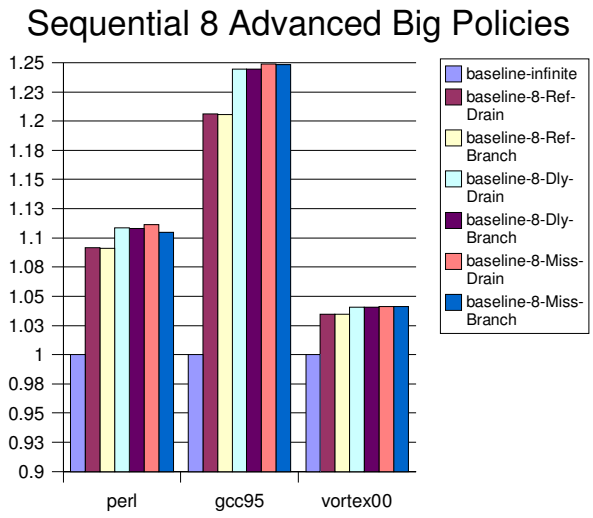


(a) Big Window

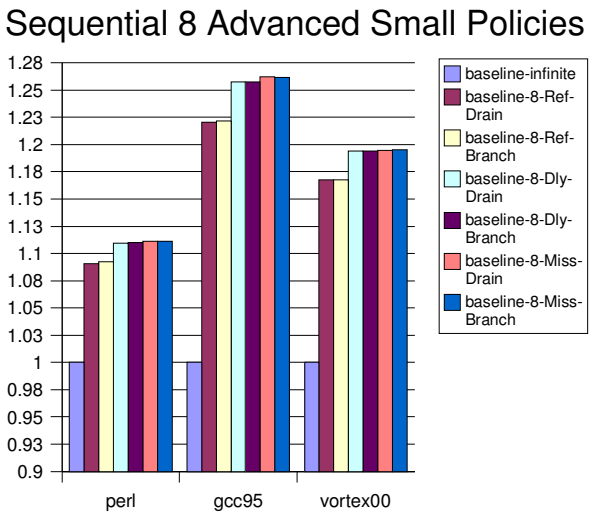


(b) Small Window

Figure 6.25: Baseline Sequential-8 Default Queue Policies Performance



(a) Big Window



(b) Small Window

Figure 6.26: Baseline Sequential-8 Advanced Queue Policies Performance

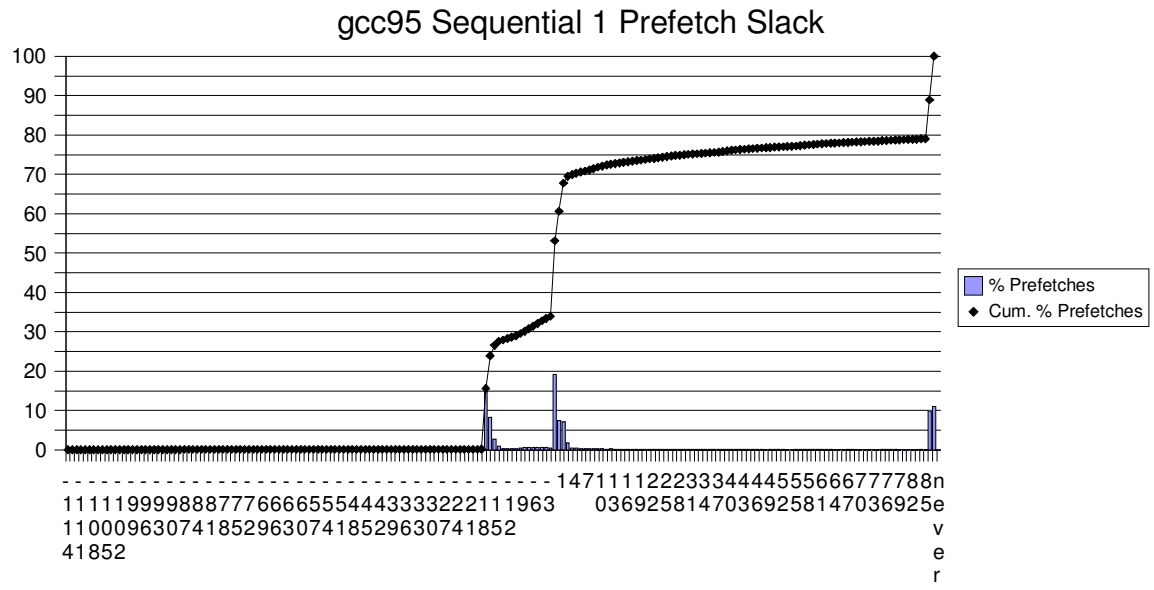
We do not present results for Sequential vs. Parallel prefetch request generation for two reasons. The additional hardware for parallel address generation, as indicated by figure 6.3(b), is negligible. Furthermore, our initial experiments with the Sequential generation policy produced extremely poor results. This most likely is due to our use of an infinite prefetch request queue (see section 6.5.1). We observed that the queueing delay involved in generating the addresses for every desired prefetch was simply too much and prefetches lost any timeliness they may have had. Therefore, we do not consider the Sequential generation policy further.

The data shows that the queueing strategy (Default vs. Advanced) has almost no impact on performance. This is not surprising because the only differences between the strategies is the request generation mechanism and the ability of the Advanced queue to clear itself when a new sequence is initiated by a cache miss or a branch. The termination policy makes little difference to the Advanced queue, confirming that the simple policy of letting prefetches drain from the queue is adequate.

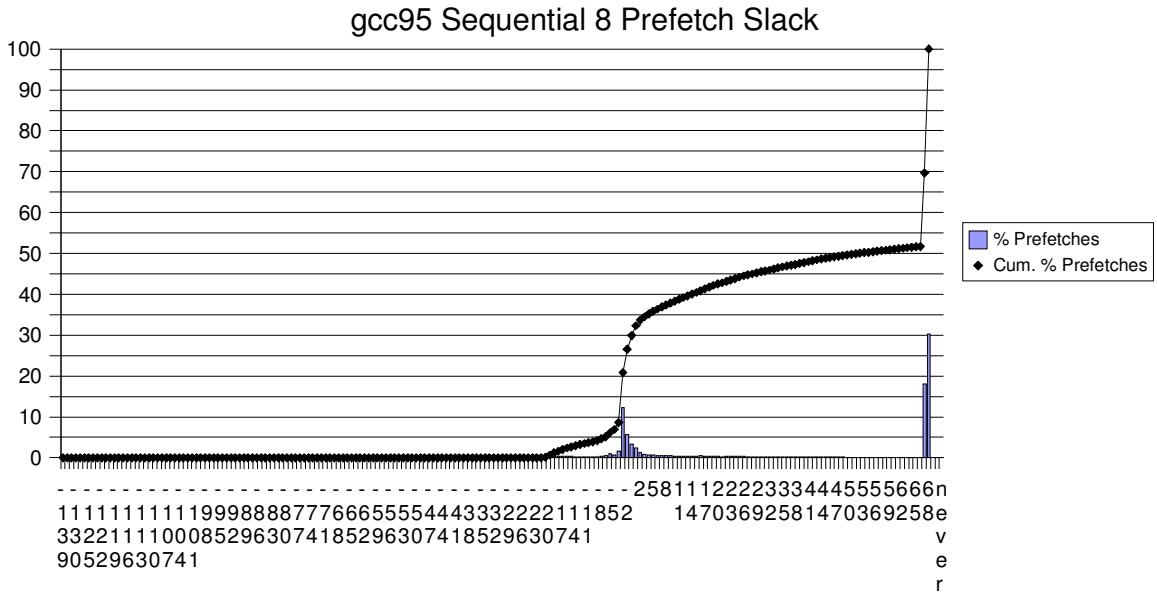
In fact the only policy which seems to make any significant difference is prefetch initiation. As indicated by previous work, initiating on every cache access is the best policy [67]. This is true even for long-distance sequential prefetching, a result not previously observed. The filtering mechanism compensates for potential cache pollution introduced by such aggressive prefetching.

These results lead us to select the following sequential prefetching strategy for the remaining experiments: initiate on Reference, generate addresses in Parallel, use the Default prefetch request queue and let the queue Drain when new prefetch sequences are initiated. This is a very simple yet highly effective strategy.

Figure 6.27 shows the prefetch slack for sequential prefetching. We define *prefetch slack*



(a) One Line



(b) Eight Line

Figure 6.27: Sequential Prefetching Slack

as the difference in cycles between the time a prefetched block is reference by a demand access and the time the prefetched block arrives from lower levels of the memory hierarchy. Negative values indicate late prefetches while a value of zero indicates the prefetch was initiated at just the right moment. The slack is calculated using the best performing policies for sequential prefetching and later, for each scheme examined.

Overall, one-line sequential prefetching has very poor timeliness. 34% of the prefetches are late. 11% of the prefetches are never accessed and are likely kicked out of the cache before they are able to become useful. The situation improved dramatically with eight-line sequential prefetching. Only 9% of the prefetches are late, though 30% are unused. The increased amount of useless prefetches is expected but does not appear to degrade performance dramatically based on the filtering results of figure 6.22.

Architectural Impact

The above policies were arrived at through experimentation on a very aggressive machine model: eight-wide instruction issue, a very large instruction window, 4-ported instruction cache and a wide data bus between the primary and secondary caches. While we maintain this aggressive model for most of the remaining experiments, we wish to explore the effect of alternative architectures to gauge the performance of prefetching when machine resources are reduced. To that end, we ran the sequential prefetching experiments using the above polices while individually restricting the window size, issue width, number of cache ports and the bandwidth between the primary and secondary caches.

Sequential prefetching results for a small window size are already provided above so we do not reproduce them here. Figure 6.28 shows the relative slowdown of eight-line sequential prefetching compared to an infinite cache machine when the number of cache

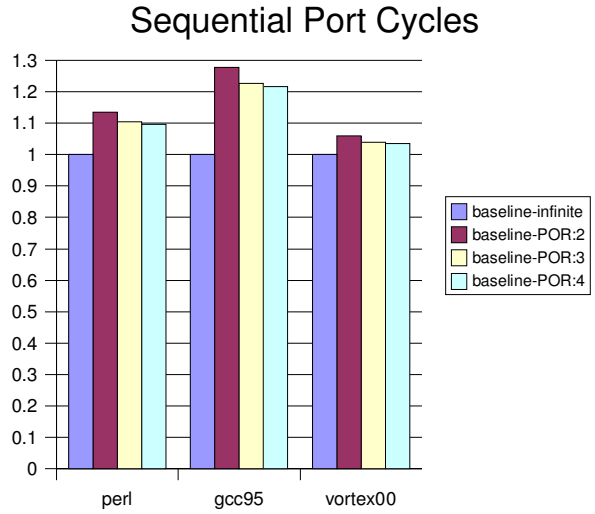
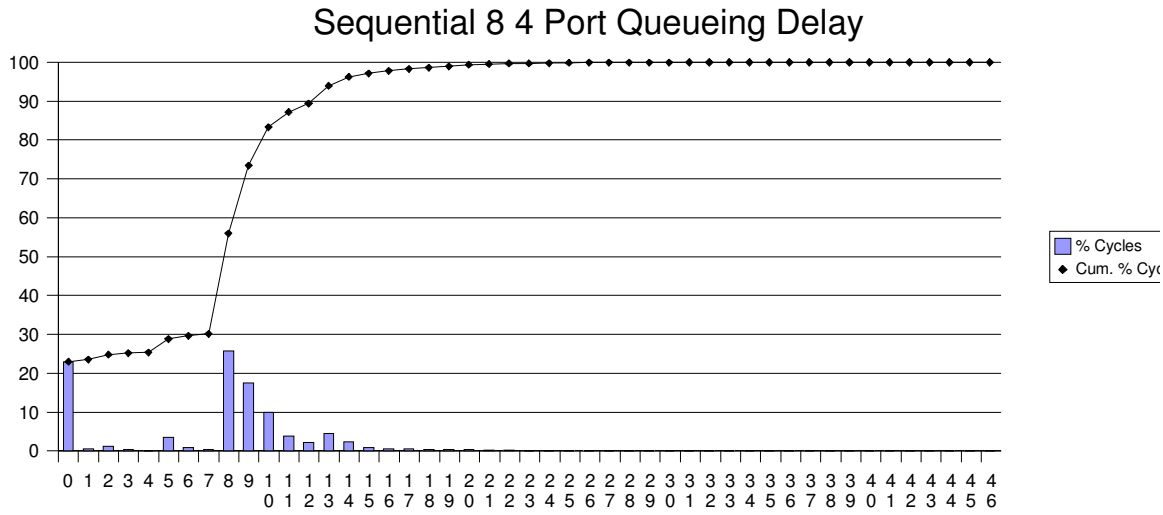


Figure 6.28: Baseline Sequential-8 Ports Performance

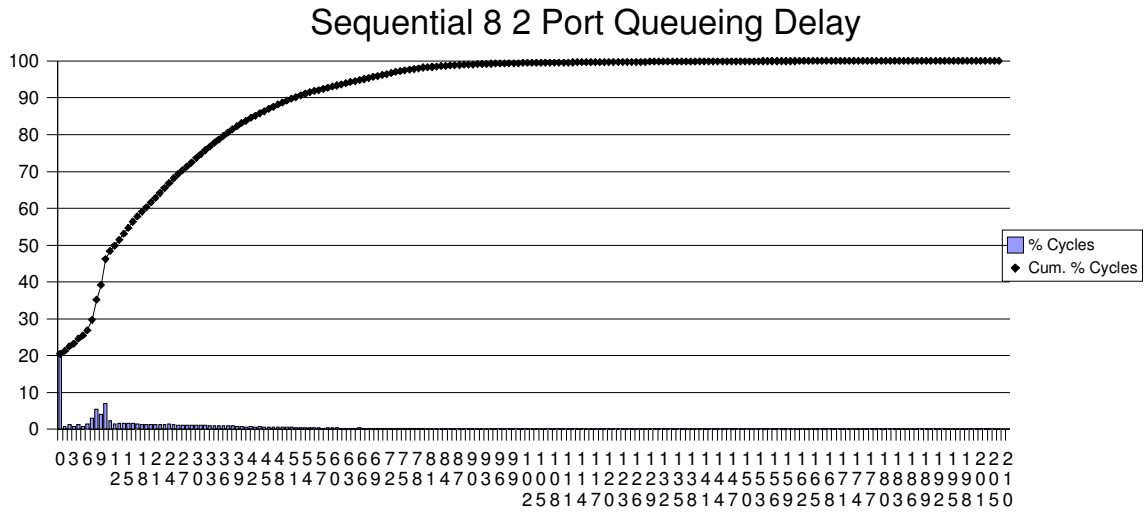
ports is restricted. Surprisingly, even restricting the number of ports to 2 only impacts performance by 6% in the worst case.

Figure 6.29 shows the Default queueing delay for sequential-8 prefetching with four cache ports and two cache ports running the vortex benchmark. The figure shows the number of cycles in which the prefetch queue holds the x-axis specified number of prefetches. Larger numbers means there is more backlog in the queue, which reduces prefetch timeliness. Both machines spend between 20% and 25% of the time with an empty queue. With the four-ported machine, 90% of the cycles are spent with 13 or fewer prefetch requests in the queue. For the two-ported machine, the 90% point is at 53, indicating a much more severe backlog of prefetch requests. On a finite-queue machine many of these requests would have been dropped due to lack of queue space. Even so, figure 6.28 indicates that it is not a serious problem.

Figure 6.30 shows the queueing delay on an four-ported machine using the Advanced queue. A significant reduction in queue length is apparent. Almost half of the cycles are spent with the queue empty. 90% of the cycles are spent with eight or fewer requests in the



(a) Four Ports



(b) Two Ports

Figure 6.29: Baseline Sequential-8 Ports Default Queueing Delay

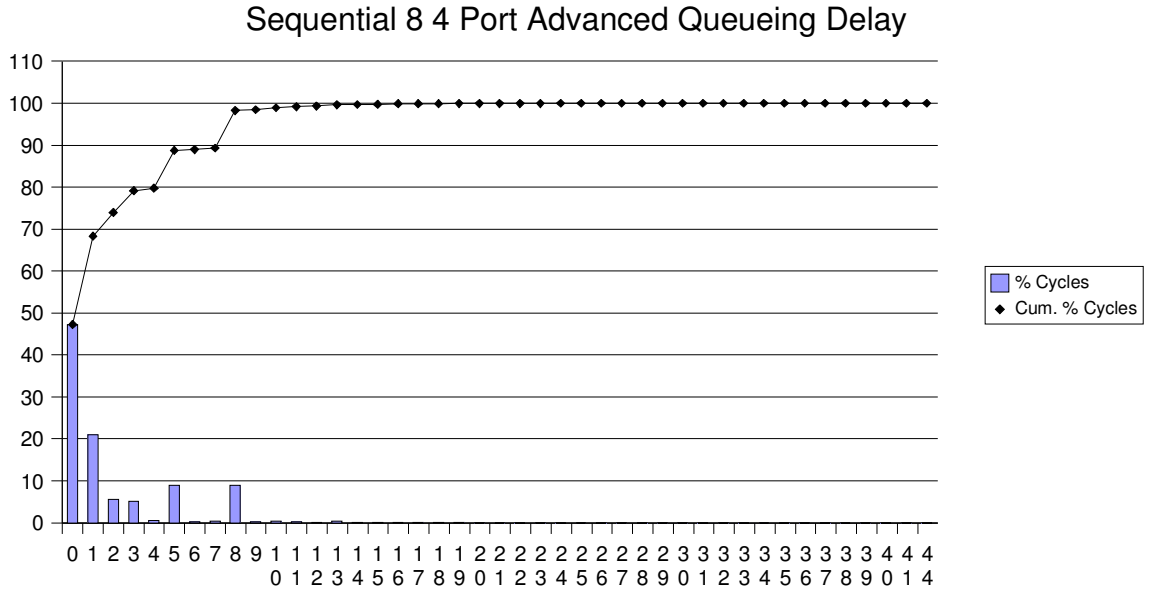


Figure 6.30: Baseline Sequential-8 Ports Advanced Queueing Delay

queue. In addition, we see a spike at five requests as well. This is to be expected because four of the prefetch requests can run in parallel and the prefetch controller is smart enough not to regenerate those requests on the next demand fetch. One request will be generated to cover the line after the last prefetch sequence, leaving five requests in the queue. Though the Advanced queue does not impact machine performance significantly, it can be used to reduce the complexity of the prefetch hardware.

Figure 6.31 shows the performance of sequential prefetching when the primary to secondary cache bandwidth is restricted. The first bar shows the infinite cache performance, the second performance with an eight-byte (quarter cache line) bus between the L1 and L2 caches and the third performance with a 32-byte bus. Performance is degraded by 12% in the worst case. This contrasts sharply with the results obtained by Luk and Mowry, who report less than 2% performance degradation on gcc95 using eight-line sequential prefetching [45].

Figure 6.32 shows the performance of eight-line sequential prefetching when the machine

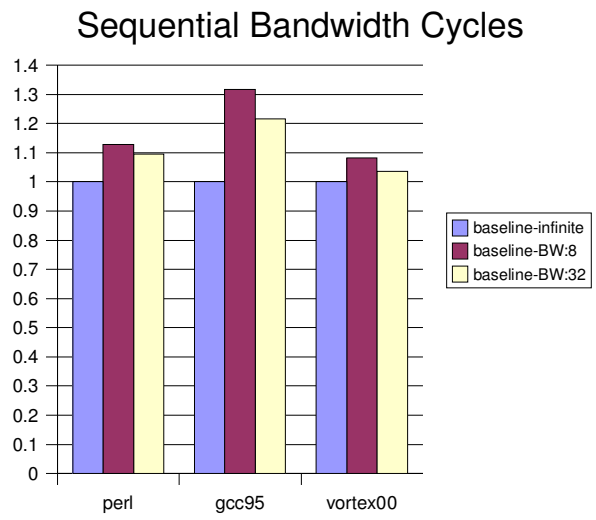


Figure 6.31: Baseline Sequential-8 Bandwidth Performance

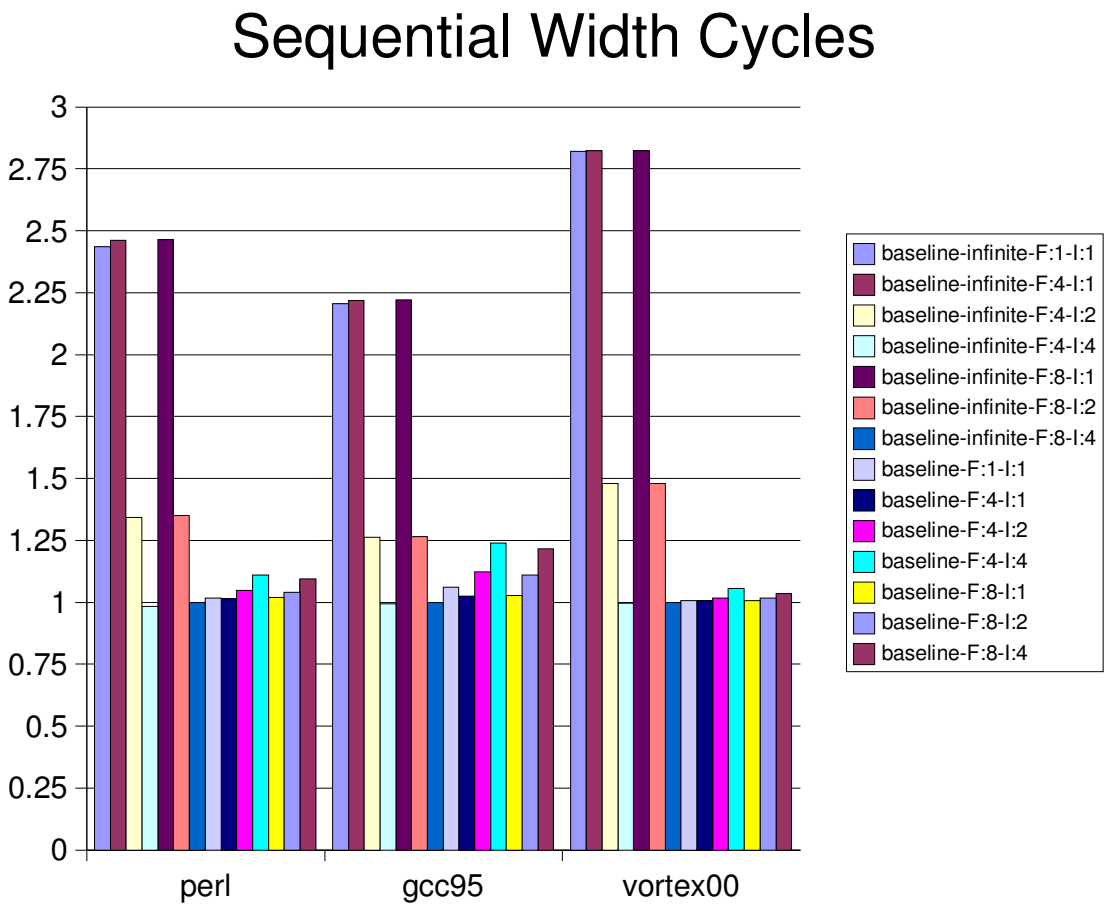


Figure 6.32: Baseline Sequential-8 Width Performance

width is restricted. The seventh bar is our baseline configuration: an infinite instruction cache machine that can fetch eight instructions per cycle and issue four of them to the function units per cycle. To the left we show the relative performance of the infinite-cache machine as the fetch and issue widths are reduced. Going left we first reduce the issue width to four and then to one, then we reduce the fetch width to four, measuring performance with issue widths of four, two and one and at the far left we present the relative performance of a one-wide fetch and issue machine. To the right of the baseline configuration we present the performance of sequential-eight prefetching relative to the infinite-cache machine of the same width. That is, the eighth bar shows the performance of sequential prefetching on a one-wide machine relative to the performance of the infinite-cache one-wide machine. The ninth bar shows the performance of sequential prefetching on a fetch-four, issue-one machine relative to the infinite cache performance of the fetch-four, issue-one machine, and so on.

Some interesting results are apparent. For the infinite cache configurations, The fetch-four, issue-four machine actually slightly outperforms the fetch-eight, issue-four machine on gcc95 and perl. We attribute this to branch prediction. On the fetch-eight, issue-four machine, gcc95 sees a slightly lower BTB hit rate than on the fetch-four, issue-four machine. Direction and return address stack prediction is also higher on the narrower machine. The same trend is observed for the perl benchmark. This is most likely due to the reduced amount of speculation on the machine that fetches fewer instructions per cycle. Because fewer branches will be seen per cycle on average, the machine has a bit more time to resolve earlier branches and update the branch history, leading to a somewhat more accurate prediction. This effect was noted by Skadron, *et al.* in their study of branch prediction, window size and cache size tradeoffs [75].

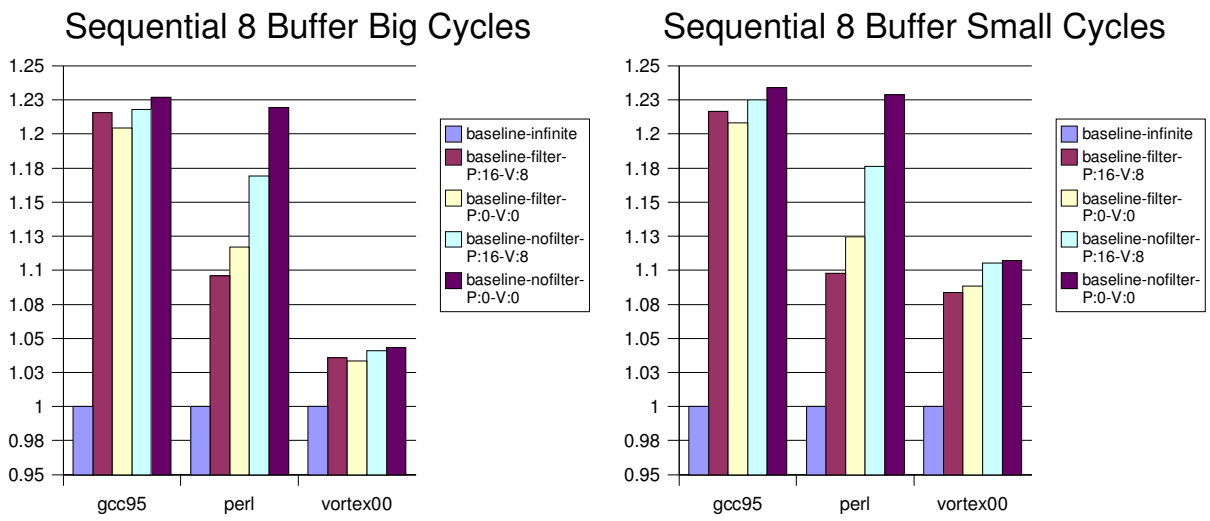
Sequential prefetching achieves better relative performance on the narrower machines.

A narrower machine sees a smaller penalty from instruction cache misses because it does not fetch down the program as quickly as a wider machine. On the one-wide machine, it will take eight cycles to consume a cache line fetch while the eight-wide fetch machine will consume an entire cache line in one cycle. Any prefetches initiated on a demand fetch will have seven additional cycles to resolve before the next demand fetch requires the target. This is over half of the primary-to-secondary latency. Therefore two effects are at play here. The machine sees a smaller finite cache penalty because it is not consuming instructions as quickly as the wider machine. Therefore sequential prefetching does not have to make up as much ground. Furthermore, the additional slack afforded by the reduced instruction rate gives prefetches more time to resolve and hide additional cache latency.

The same reasoning explains why relative performance of sequential prefetching on the fetch-four machines is slightly worse than on the fetch-eight machines. Because the infinite cache fetch-four performance is higher than on the infinite cache fetch-eight performance, prefetching has to make up more ground. The fetch-four machine also consumes instructions more quickly, meaning the prefetcher has less time to resolve targets and hide cache latency.

The prefetch buffer and victim cache help to reduce the effects of cache pollution by the prefetch engine. To quantify this effect, figure 6.33 shows the performance of eight-line sequential prefetching without these structures. For each benchmark, the first bar is the relative infinite cache performance. It is always one. The next two bars show the performance of sequential prefetching with hardware filtering, first using the additional buffers and then without. The last two bars show the performance when the hardware filter is disabled.

Performance is only degraded 1%-2% for gcc and vortex, but perl suffers a bit more. Its performance is reduced by 12% on the large window machine when the buffers and hardware



(a) Large Window

(b) Small Window

Figure 6.33: Baseline Sequential-8 No Buffer Performance

filter are removed. Small window performance suffers 13%. On perl the effect of filtering is more pronounced when the extra buffers are removed but the other benchmarks see little effect. It is interesting to note that gcc95 performance improves when the prefetch buffer and victim cache are removed if hardware filtering is in place. This could be due to any number of complex interactions between the cache, prefetcher and instruction scheduler.

6.6.2 Cooperative Prefetching Results

In this section we present results for the Cooperative Prefetching scheme. We begin by quantifying the overhead of instruction prefetching followed by a policy study. We then present results for varying architectures.

Overhead

There are two main components to the overhead of software instruction prefetching. The first is the static code bloat caused by the insertion of the prefetch instructions. Such

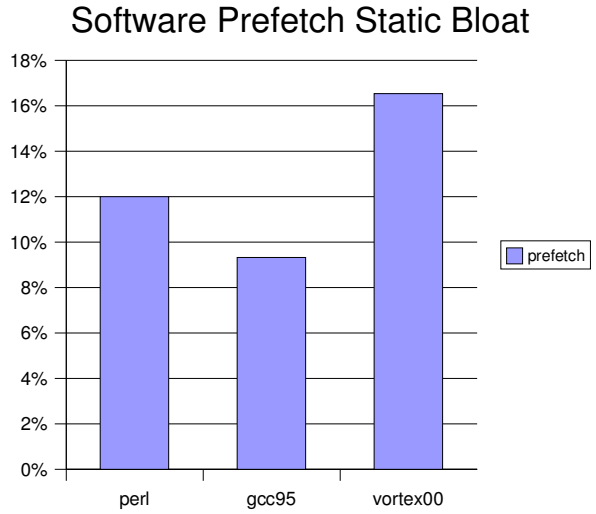


Figure 6.34: Static Prefetch Bloat

bloat may reduce instruction cache effectiveness by increasing the program's memory footprint. The second is the overhead involved in fetching, decoding and executing the prefetch instructions.

To quantify the static code bloat, we measured the static number of instructions in the program before and after prefetch scheduling. Because our compiler cannot yet instrument system libraries, these numbers do not include instructions from these libraries. Thus the code bloat indicated is over the compiler-visible program. Figure 6.34 indicates that the static overhead is a bit higher than earlier reported results [45]. The vortex benchmark suffers the most overhead at about 16.5% as compared to 11% reported by Luk and Mowry. Both gcc and perl are within 2.1% of the earlier results. It is not surprising that these numbers would change owing to the different instruction sets and compilers used. We did use a smaller cache (8K vs. 32K) in our studies, but as mentioned earlier, the grossly overinflated size estimates for procedures makes this nearly irrelevant. Such estimates would not have fit those procedures into a 32K cache. However, our numbers are not drastically different than the earlier results, lending some confidence that we have not radically altered

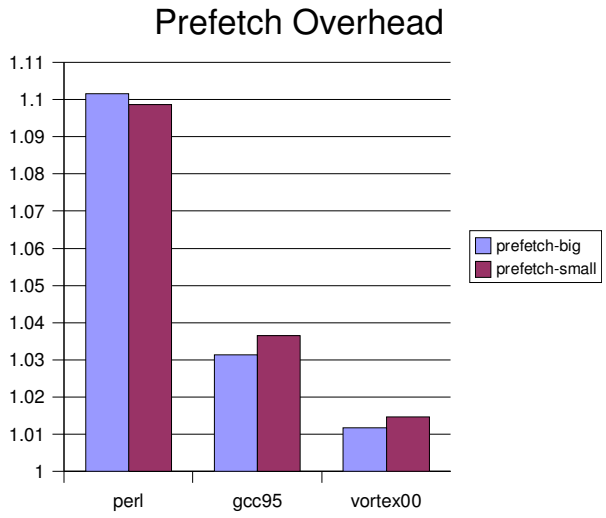


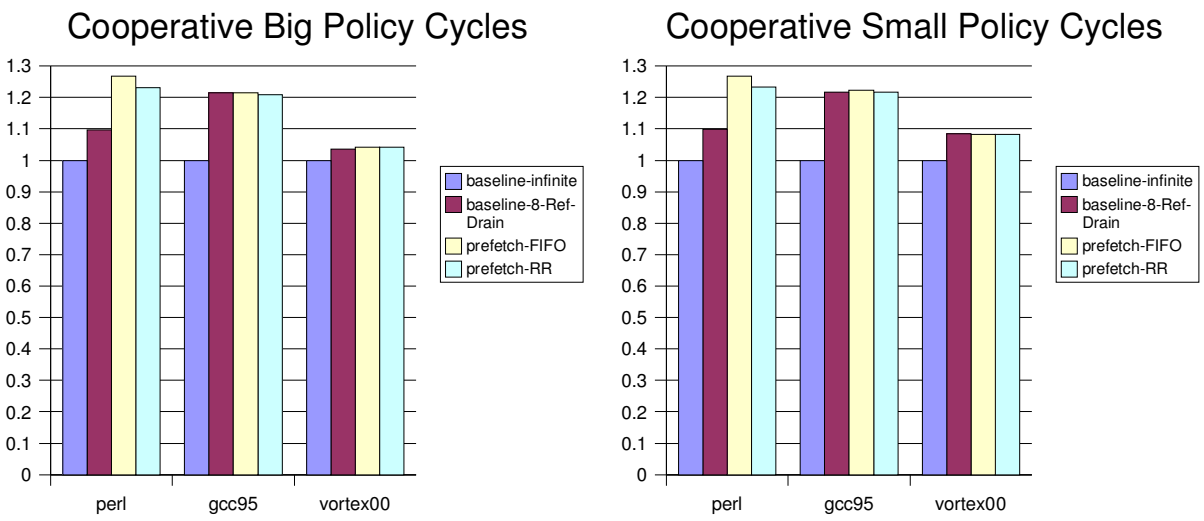
Figure 6.35: Dynamic Prefetch Overhead

the scheduling algorithm in some way.

From figure 6.35 we see that the dynamic overhead is fairly small except for the perl benchmark. The performance penalty on a machine with eight-line sequential prefetching is 1% for vortex and 3%-4% for gcc depending on window size. The 10% overhead experienced by perl is going to be very difficult to overcome. After investigating the causes of this extreme overhead, we noted that a very large number of prefetches was placed in the `eval` procedure, a highly unstructured piece of code with several large `switch` statements. It is a routine called from many places in the benchmark and we believe that this is the primary source of overhead. It may be that our heuristics simply do not perform well on this particular piece of code.

Policies

We present the performance of Cooperative Prefetching with the FIFO and RoundRobin scheduling policies in figure 6.36. As expected, Cooperative Prefetching reduces performance for perl. For the most part Cooperative Prefetching recovers its overhead and changes



(a) Big Window

(b) Small Window

Figure 6.36: Cooperative Prefetching Policies Performance

performance only very slightly. The only configuration where we observe a performance improvement is running the gcc95 benchmark on a large window machine with RoundRobin scheduling. Given our widely differing baseline results above, it is not surprising that we do not observe the performance improvement of Cooperative Prefetching reported by Luk and Mowry. Unfortunately, without more information about the experimental setup used, it is difficult to conduct further investigation.

Prefetch slack measurements for Cooperative Prefetching appear in figure 6.37. Timeliness is about the same as for eight-line sequential prefetching. 8% of the prefetches are late while 39% are never used. It appears as though the distant prefetches performed by the prefetch instructions are not being used. Inspection of the miss traces for the gcc95 confirmed this. One address in particular accounted for a majority of the unprefetched misses. The compiler did in fact schedule an instruction prefetch for this block but it was placed very near the top of the routine and is likely kicked out of the cache before it is ever

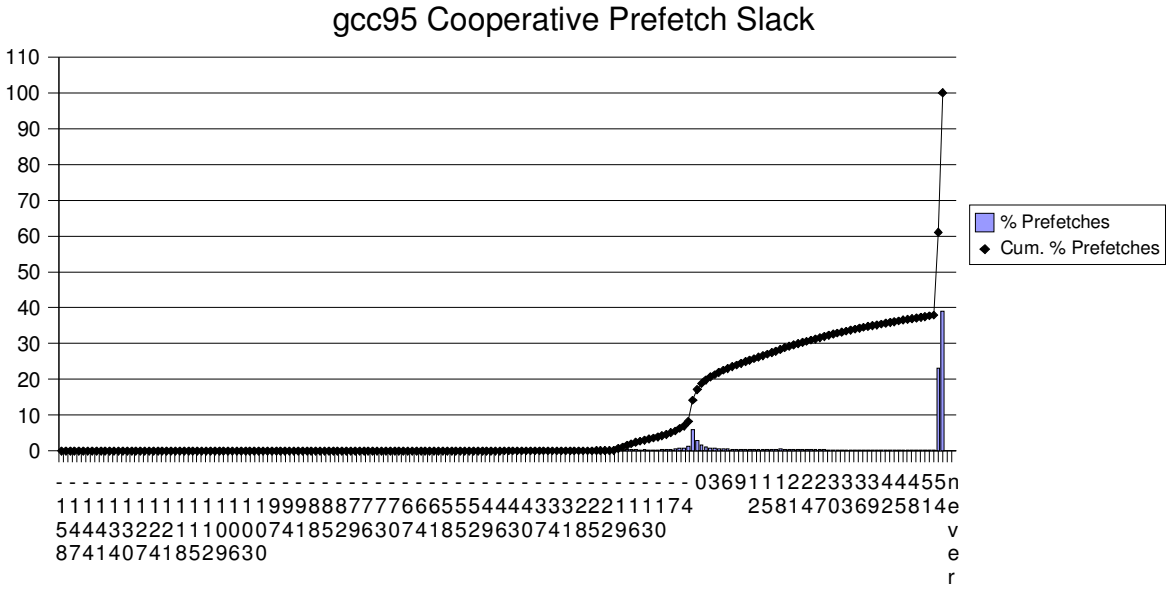


Figure 6.37: Cooperative Prefetching Slack

used. It may be that our scheduling filters have enough variation over those used by Luk and Mowry to account for the performance differences.

Architectural Impact

Even so, it is possible that architectural changes may alter the picture. Therefore, we ran experiments varying machine architectural parameters to see if there are other design points where Cooperative Prefetching might be viable.

The effect of reducing the available cache ports for Cooperative Prefetching is illustrated in figure 6.38. It is clear that cache ports are much more important for Cooperative Prefetching than for sequential prefetching alone. While sequential prefetching performance is only reduced at most 6%, perl suffers a 13% degradation with a two-ported instruction cache and gcc95 experiences an 11% degradation on the same machine. Except for the gcc success noted earlier, Cooperative Prefetching performs worse on every port configuration examined.

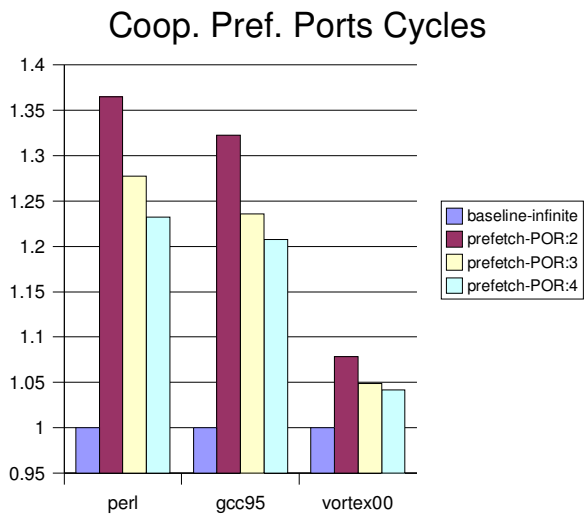


Figure 6.38: Cooperative Prefetching Ports Performance

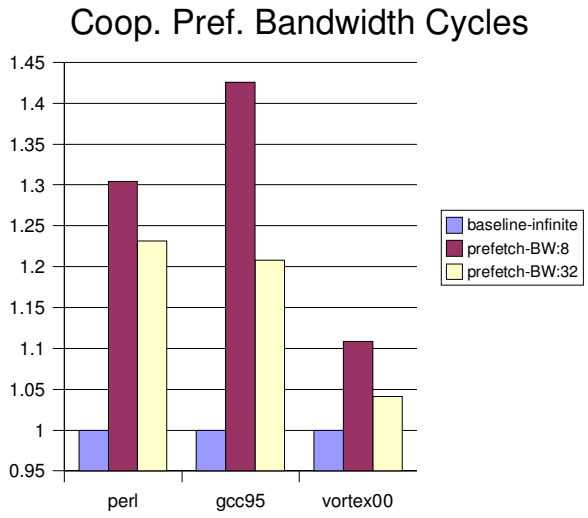


Figure 6.39: Cooperative Prefetching Bandwidth Performance

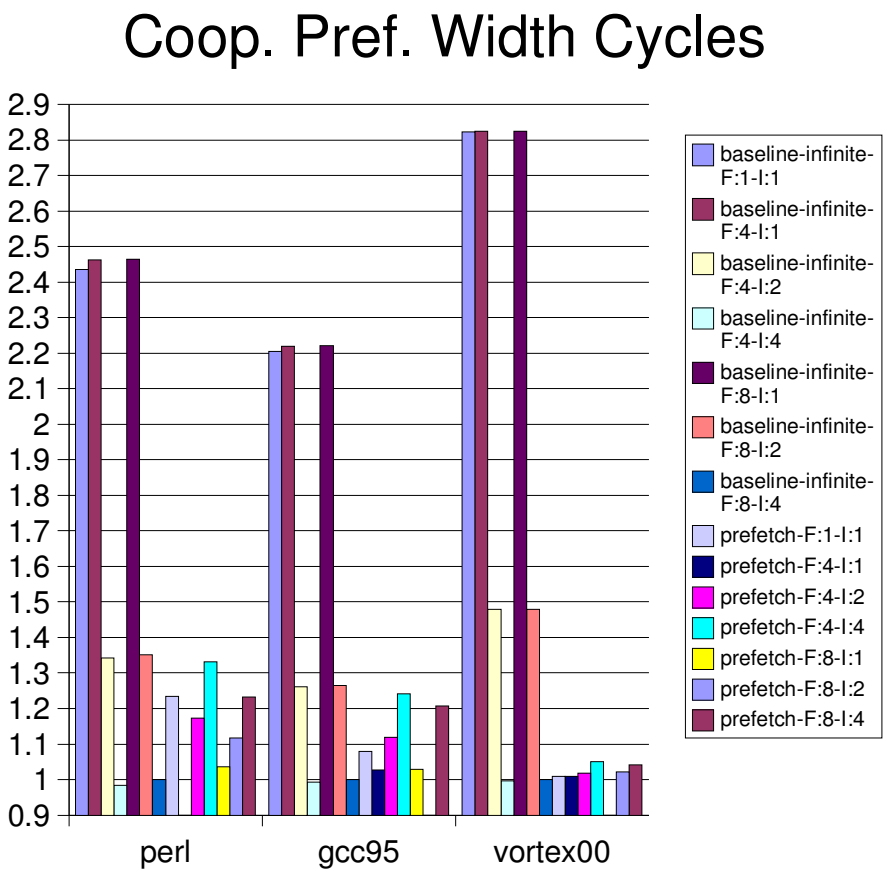


Figure 6.40: Cooperative Prefetching Width Performance

Figure 6.39 indicates that Cooperative Prefetching suffers even worse under limited cache bandwidth. This again contrasts sharply with previously published results. The gcc benchmark suffers a whopping 22% performance degradation with the narrow data bus. Sequential prefetching easily outperforms Cooperative Prefetching on bandwidth-limited machines.

Cooperative Prefetching performance follows sequential prefetching performance with decreasing machine width, as shown in figure 6.40. The presentation schema is the same as in figure 6.32. The perl benchmark suffers slightly less worse under Cooperative Prefetching than sequential prefetching as issue width increases. While sequential prefetching alone roughly doubles in relative slowdown as issue width increases from four to eight, Cooperative

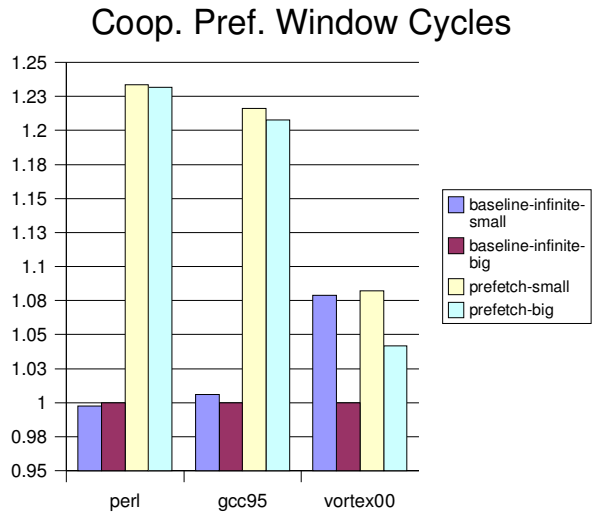


Figure 6.41: Cooperative Prefetching Window Performance

Prefetching suffers slowdowns of under 2x. The relative penalty moving from two to four issue is much worse. It is important to remember that in all cases Cooperative Prefetching still has worse absolute performance than sequential prefetching on the same architecture.

Finally, figure 6.41 shows how Cooperative Prefetching performance changes as the window size is altered. Unlike with sequential prefetching, Cooperative Prefetching seems to benefit slightly more from a large instruction window. This is somewhat surprising because software instruction prefetches do not occupy any window space. It may be that the large window is making the distant prefetches of Cooperative Prefetching relatively more important because their targets are reached more quickly than in a machine that may stall with a full instruction window.

6.6.3 BHGP Results

Our initial experiments with BHGP produced puzzling results. We did not observe the published result that BHGP outperforms one-line sequential prefetching. We speculated that the processor pipeline might play a detrimental role by decreasing prefetch timeliness.

Br_5	E
Br_4	D
Br_3	C
Br_2	B
Br_1	A

Figure 6.42: Example BHGP State

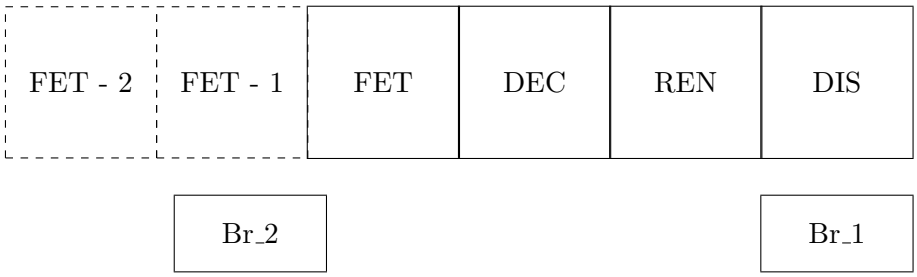


Figure 6.43: Lower-Bound Pipeline Penalty

Because branches, instruction prefetches and annotated instructions are serviced at the end of decode, their effective distance in terms of the number of branches ahead of which they are serviced is decreased. The M5 simulator calculates the "branch fetch rate" (BFR) of execution as the number of branches fetched on average per cycle. Simulation results for vortex00 with 8K of instruction cache and a one-line sequential prefetcher show a branch fetch rate of 0.28, meaning that on average, a new branch is fetched just over every three cycles.

BHGP has a five-entry BHQ, meaning it is attempting to prefetch five branches ahead of the "current" branch. Assume the BHGP prefetch table has the state shown in figure 6.42 and we are about to execute the branch sequence (Br_1, Br_2, Br_3, Br_4, Br_5). Figure 6.43 presents a pipeline model of the machine front-end. There is a single fetch stage followed by two stages of rename/decode and ending in the dispatch stage. Our experiments above assume that branches are sent to the BHGP prefetcher in the dispatch stage, as shown

by the presence of Br_1 in that stage. Looking back through the pipeline we see branches interspersed at several points according to the branch fetch rate. To the left of the pipeline are "future" pipeline stages containing branches we will be fetching in the near future (assuming no cache misses).

The five-entry BHQ is supposed to allow the prefetcher to look ahead five branches and prefetch the target for the fifth-most "future" branch. From figure 6.43 we can see that the effective length of the BHQ has been shortened. We are somewhere in-between fetching and decoding Br_3 when Br_1 is seen by the prefetcher. That leaves less than three branches before we will fetch the target of Br_5. Given the total number of fetch and decode pipeline stages S, the effective BHQ size can be calculated with the following equation:

$$|BHQ|_{eff} = |BHQ| - BFR \times S \tag{6.7}$$

Given the above branch fetch rate for vortex00, the effective BHQ size is 4.15.

When the front-end of the pipeline is stalled for a cache miss or some other reason[5], no prefetching will be performed and no branches will be in the decode pipeline, so the measured branch fetch rate gives a lower bound on the pipeline penalty for transition prefetching. The simulator also keeps track of the number of "fetch chances," (FC) those cycles in which the front-end can access the instruction cache. Using the count of the total number of branches fetched (BF) during execution, we can get an upper bound on the branch fetch rate with the following formula:

$$BFR_{ideal} = \frac{BF}{FC} \tag{6.8}$$

[5]Such as a full instruction window or branch mispredict recovery

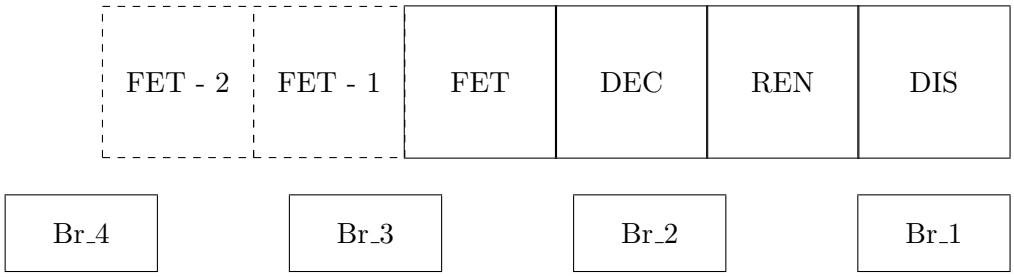


Figure 6.44: Upper-Bound Pipeline Penalty

$$|BHQ|_{eff} = |BHQ| - BFR_{ideal} \times S \tag{6.9}$$

The ideal branch fetch rate for vortex00 using an 8k cache and one-line sequential prefetching is 0.56, leading to the pipeline penalty diagram of figure 6.44. The lower-bound effective BHQ size is given by equation 6.9 and is 3.31 for vortex00. The true penalty for transition prefetching is somewhere in-between these two extremes.

This analysis also applies to the Cooperative Prefetching and CHGP algorithms. In those algorithms, the BHQ is replaced by a compiler scheduling algorithm. The algorithm we used schedules prefetches at least 20 dynamic instructions ahead of their targets. The pipeline penalty effectively shortens this distance.

The measure the effects of the pipeline penalty we ran the policy experiments for Cooperative Prefetching, BHGP and CHGP assuming that branches, instruction prefetches and annotated instructions could be handled in fetch, the earliest possible time any such processing could be done. Note that for BHGP and CHGP there is still a one cycle delay between the prefetcher access and the cache access due to the latency of the prefetch table lookup.

While these changes improved BHGP somewhat, it still did not outperform sequential prefetching. We obtained a copy of the BHGP simulator used by Srinivasan, *et al.* and

compared it to our implementation. We identified the following differences:

1. If the BHQ is not full, no prefetch associations are added to the prefetch table. We added all such associations.

2. If a branch already exists in the BHQ, it is not added. Our simulator adds such branches to the queue.

3. A change of program sequencing not caused by an actual branch instruction updates the target block address of the most recently executed branch. In other words, if a mispredicted branch recovers before the next branch is processed, the BHGP state will be updated to reflect the correct-path target. Our model did not perform this update.

4. Due to the above update, prefetch distance is calculated by taking the difference of the target block address and the address of the next branch instruction. If this distance is zero, one line is prefetched starting at the target address. Our simulator increments a counter on each instruction cache access as implied by the published mechanism.

5. The original simulator truncated prefetch distances at nine cache lines. We performed no such truncation.

6. The original model assumed a ten-ported instruction cache. One port for a demand fetch and up to nine ports used for prefetching.

7. The original simulator maintained a binary tree of all prefetch associations ever made. The prefetch table interface is simply a timing model on this data structure. The effect is that of an infinite prefetch table in core memory cached by the prefetch

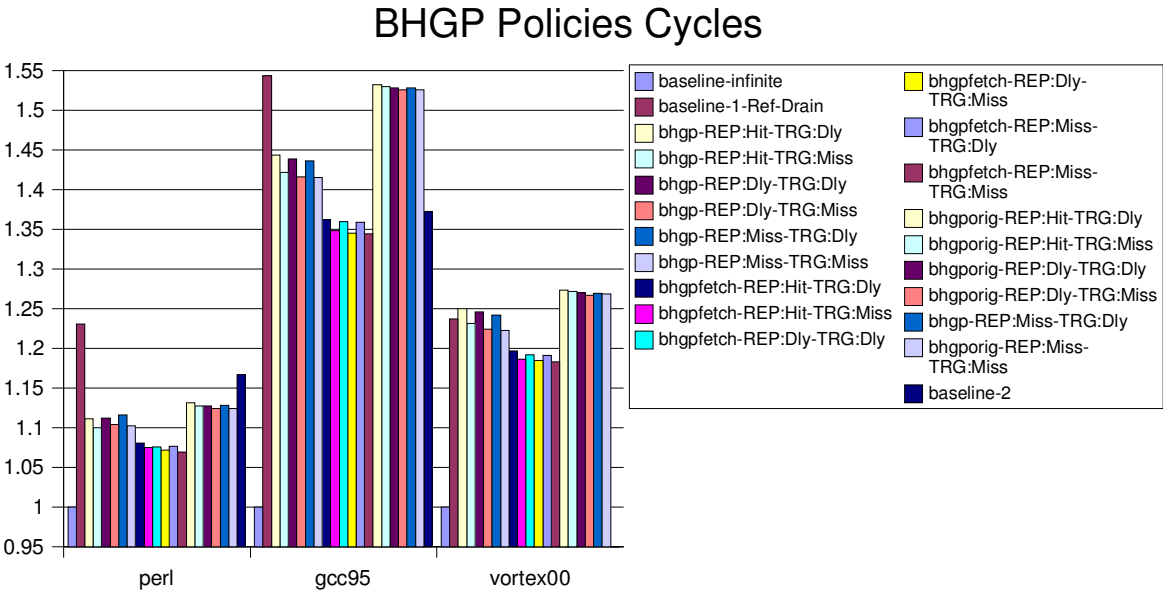


Figure 6.45: BHGP Performance

table interface. We maintained a true 4K prefetch table with the associated loss of information as implied by the original publication.

We modeled all but the last change in an additional set of experiments to determine the impact of these differences. As the results will show, they in fact hurt performance of BHGP. Our conclusion is that the last item (prefetch table size) is the determining factor. Unless otherwise specified, all of the BHGP, CHGP and Cooperative CHGP results presented below were obtained with an 8K instruction cache and 32K prefetch table. We note that this prefetch table size cannot be justified as it is four times larger than the cache itself. We performed a set of experiments to determine the prefetch table size best suited to a particular cache size. We chose 32K as an arbitrary design point that produced good results and allowed us to study other policy and architectural effects on the table-based prefetchers. We do not recommend this as a desirable design point.

Figure 6.45 shows the results of various policies for Branch History Guided Prefetching.

There are 21 bars for each benchmark. The first is the relative performance of the infinite cache architecture. It is always one. The second bar shows the performance of one-line sequential prefetching. Our goal with BHGP is to beat this target. The next six bars show the performance of BHGP when it operates at the end of instruction decode. The six bars after that show the performance when BHGP operations in fetch. The following six bars show the effects of the original source code model described above. The final bar shows the performance of two-line sequential prefetching as a benchmark against which to compare BHGP.

It is immediately apparent that the model changes to emulate the original BHGP simulator are detrimental. Performance is worse in all cases and for vortex00 it is worse than one-line sequential prefetching. Because the original published work showed an improvement for vortex, it is likely some other difference between the models has been missed.

The other general conclusion is that operation in fetch outperforms operation in decode. In general the association trigger and table replacement policies do not make much of a difference. However, overall model differences have produced very different results, most dramatically on gcc95. There is a 19% performance gap between the best-performing fetch-based BHGP and the worst-performing original-model BHGP, easily extending over the 10% performance improvement reported in the original published work.

We do not claim that the technique has no merit. In fact we are seeing good results, though the prefetch table is rather larger. Rather, we simply wish to make the point that ambiguity in the literature can result in very different conclusions depending on how that ambiguity is handled.

It is important to note that BHGP will have little hope of matching the performance of eight-line sequential prefetching. As shown in table 6.2, most of the prefetching covers

Benchmark	Avg. Length	Max. Length
perl	2.16	11
gcc95	3.35	17
vortex00	2.17	16

Table 6.2: Average BHGP Prefetch Lengths

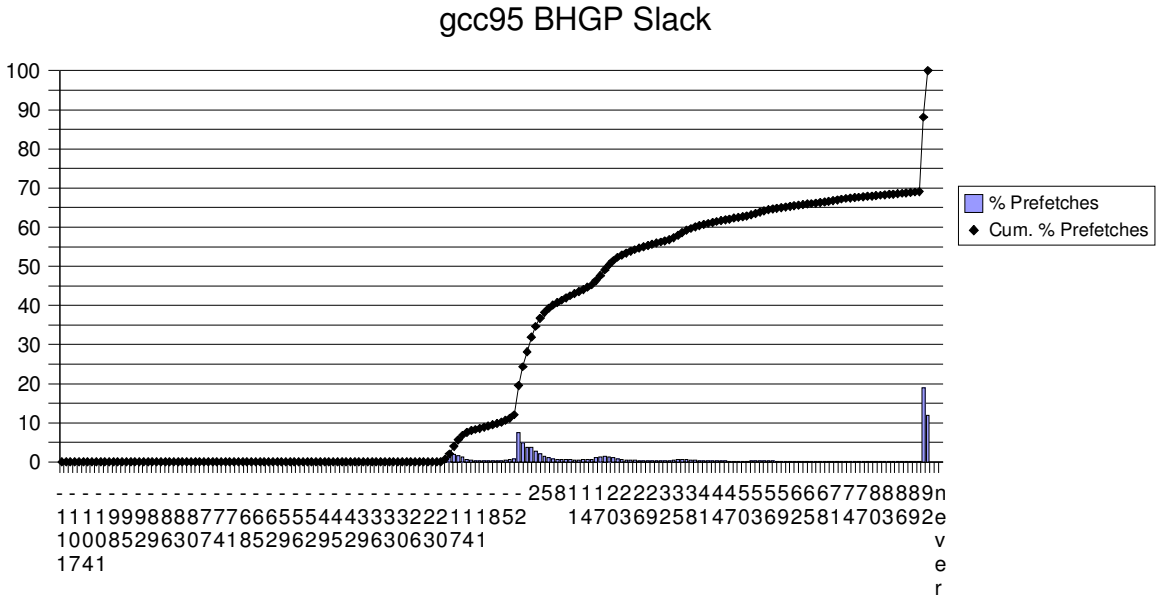


Figure 6.46: BHGP Slack

on average from 2.16 to 3.35 cache lines at any one time, meaning it is not running nearly as far ahead of the fetch engine as eight-line sequential prefetching. BHGP can potentially improve over sequential prefetching if the target branch is a function call, return or other distant branch but it appears that this is not enough to improve upon distant sequential prefetching. In fact a comparison of figures 6.45 and 6.25 show that BHGP can outperform eight-line sequential prefetching on the perl benchmark. However, it loses that advantage under realistic table sizes as will be seen later.

Figure 6.46 shows the prefetch slack for BHGP. Timeliness is much better than with one-line sequential prefetching. Only 20% of the prefetches are late though 12% are never used.

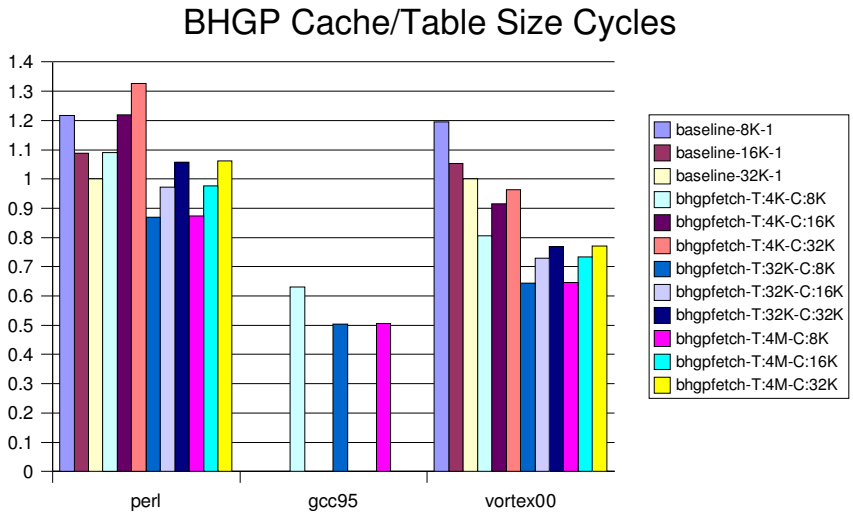


Figure 6.47: BHGP Cache/Table Size Performance

Architectural Impact

Figure 6.47 shows the tradeoffs involved between instruction cache and prefetch table size. As expected, BHGP shows less improvement as the cache size increases and as the table size decreases. A 32K table size is sufficient to show improvement over one-line sequential prefetching for all benchmarks. A 4K table size hurts performance for the perl benchmark but still shows improvement on the other benchmarks.

As one might expect, figure 6.48 indicates that BHGP is not affected at all by cache port reductions. In light of the short prefetch lengths of table 6.2 this is not at all surprising.

Decreased primary-to-secondary bandwidth has a slight affect on BHGP performance, increasing execution time by up to 10% on gcc95 as shown in figure 6.49. This is close to the results observed for eight-line sequential prefetching. It is possible that the bursty behavior of BHGP implied by the maximum values in table 6.2 may clog the bus at critical moments.

As the machine width decreases BHGP becomes relatively less worse than a perfect

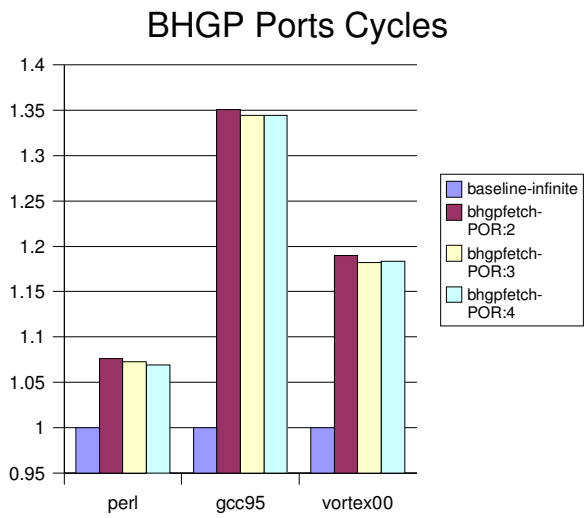


Figure 6.48: BHGP Ports Performance

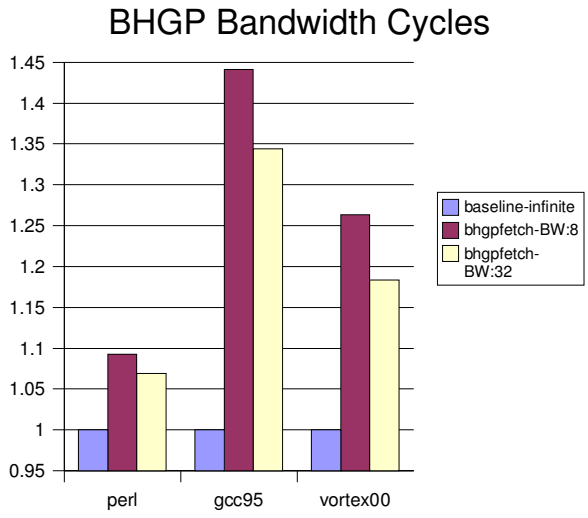


Figure 6.49: BHGP Bandwidth Performance

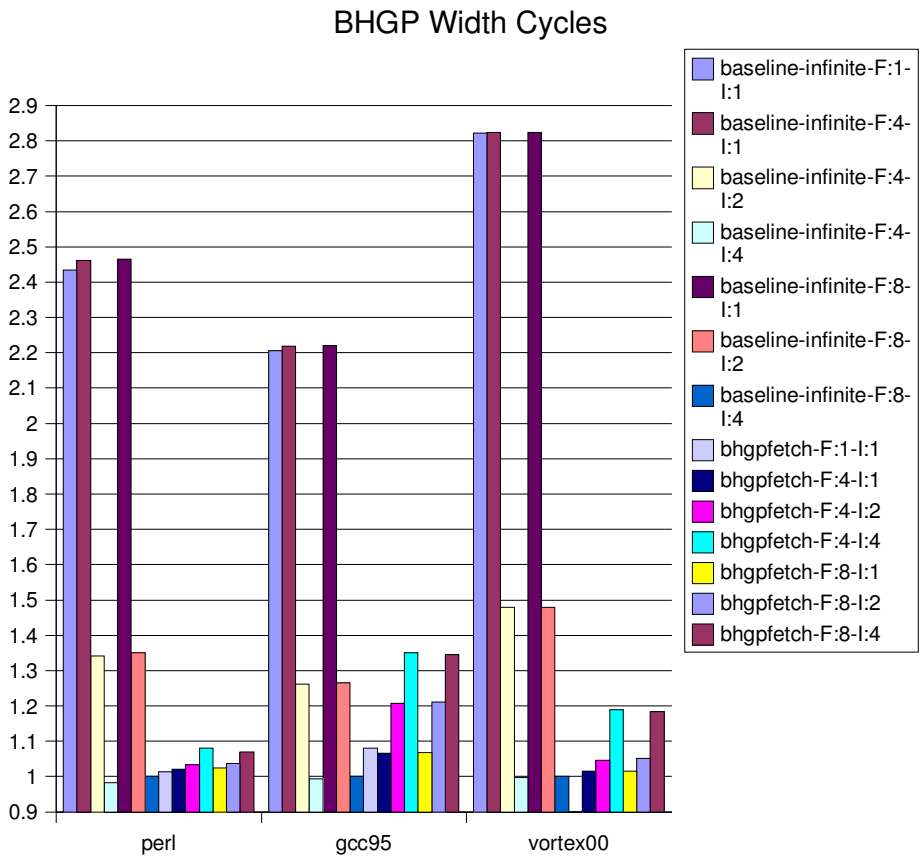


Figure 6.50: BHGP Width Performance

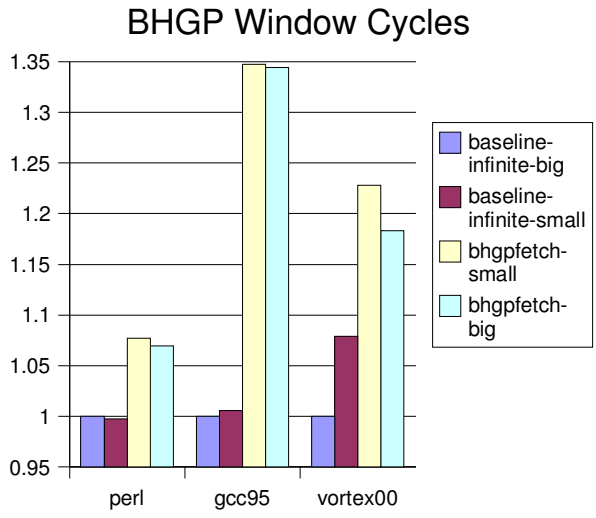


Figure 6.51: BHGP Window Performance

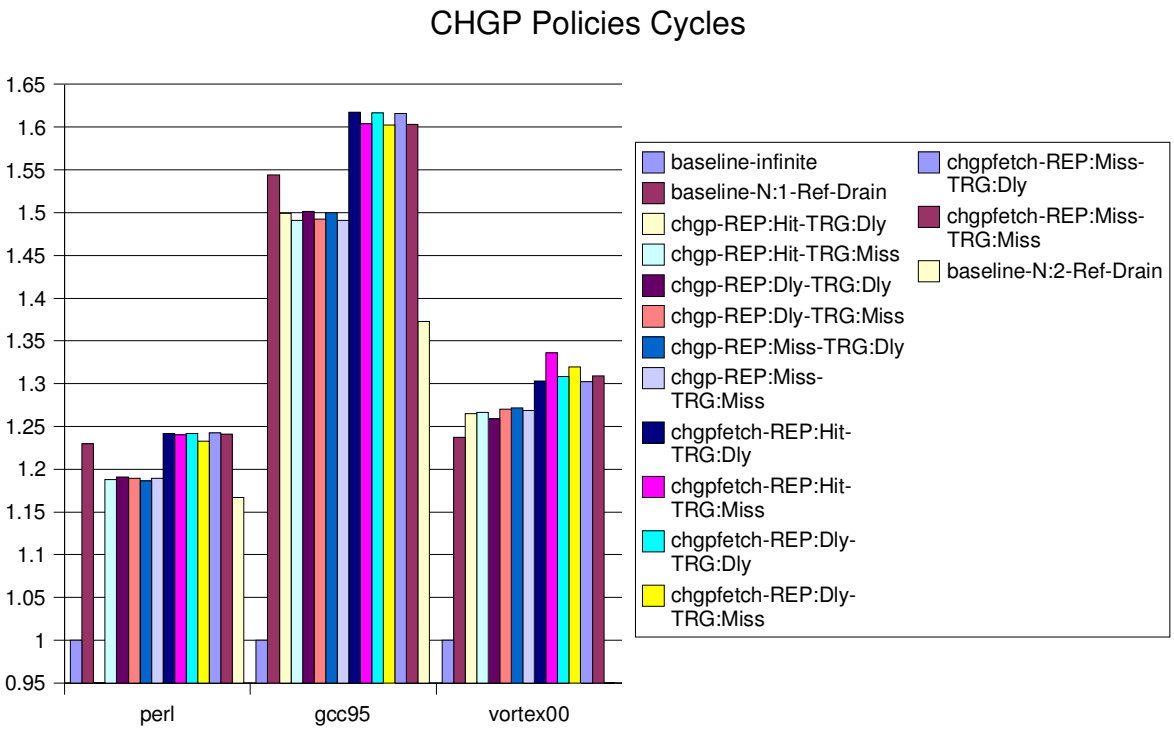


Figure 6.52: CHGP Performance

instruction cache as observed for the other schemes. We present BHGP performance on the large and small window machines in figure 6.51. The performance of vortex on the small window machine is slightly degraded while it is barely changed for the other benchmarks. As expected, BHGP does not appear to depend strongly on the machine window size.

6.6.4 CHGP Results

In figure 6.52 we illustrate the performance of Compiler Hint Guided Prefetching. Performance is relatively worse than BHGP. We attribute this to the increased placing of annotated instructions caused by the removing of the sequential path prefetch filters from the software algorithm. Because so many more instructions are annotated as potential prefetch trigger points, the prefetcher does not have time to build up state to produce distant or lengthy prefetch sequences.

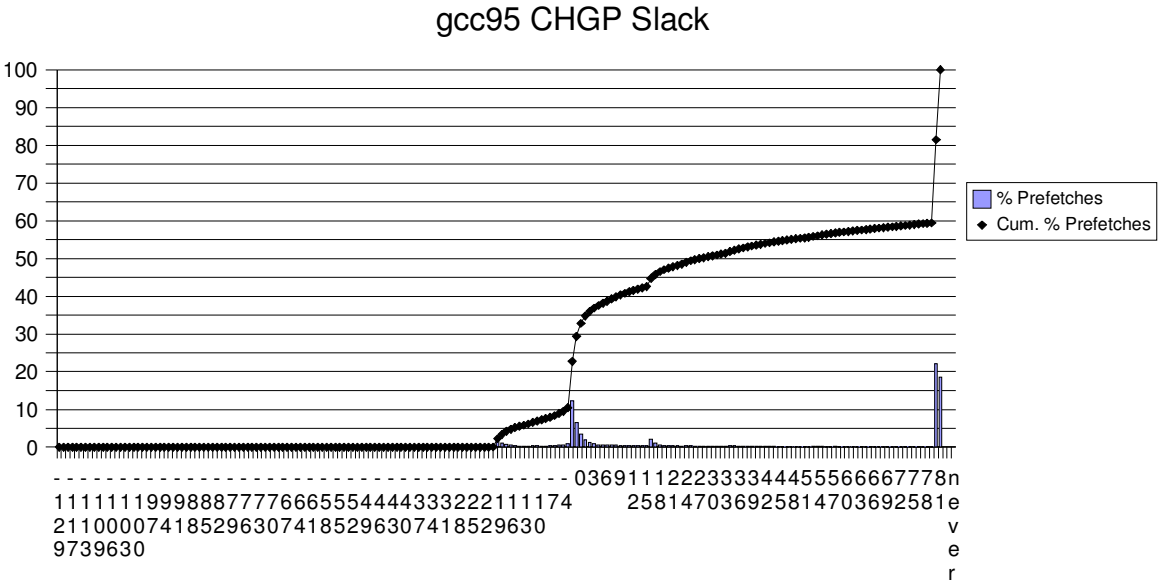


Figure 6.53: CHGP Slack

The vortex benchmark does not benefit at all from CHGP, though perl and gcc see moderate improvement. A two-line sequential prefetcher outperforms CHGP in all cases. An interesting result is that CHGP performs worse when operating in the fetch stage of the pipeline, sometimes much worse as with gcc95. It may be that operation in fetch associates cache misses that are closer to the annotated instructions resulting in decreased prefetch timeliness. The decode pipeline may provide enough space between an annotated instruction and the next miss to improve the situation. In effect, the pipeline may be taking the place of the branch queue in BHGP.

The slack information presented in figure 6.53 supports the conclusion that the multitude of annotated instructions decreases timeliness. 10% of the prefetches are late, an increase over BHGP and 18% are never used compared to 12% for BHGP. Overall the prefetch timeliness of CHGP is slightly worse than for BHGP.

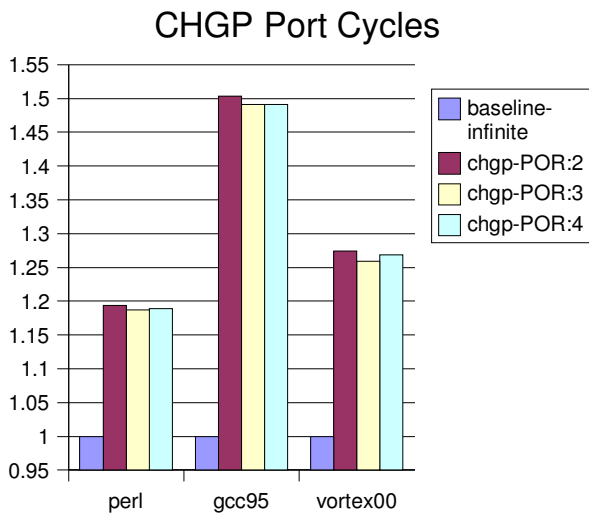


Figure 6.54: CHGP Ports Performance

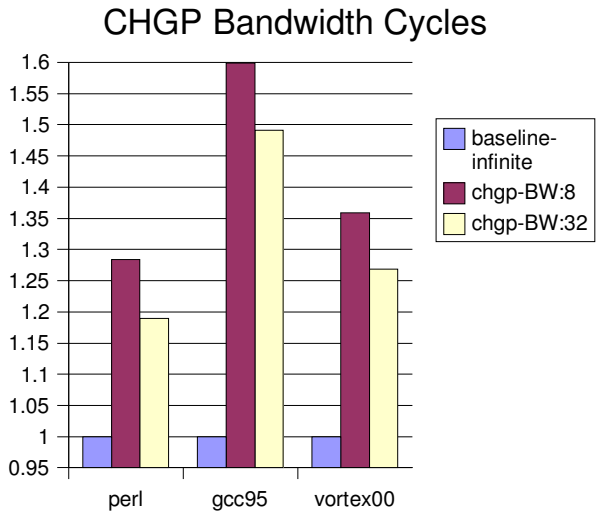


Figure 6.55: CHGP Bandwidth Performance

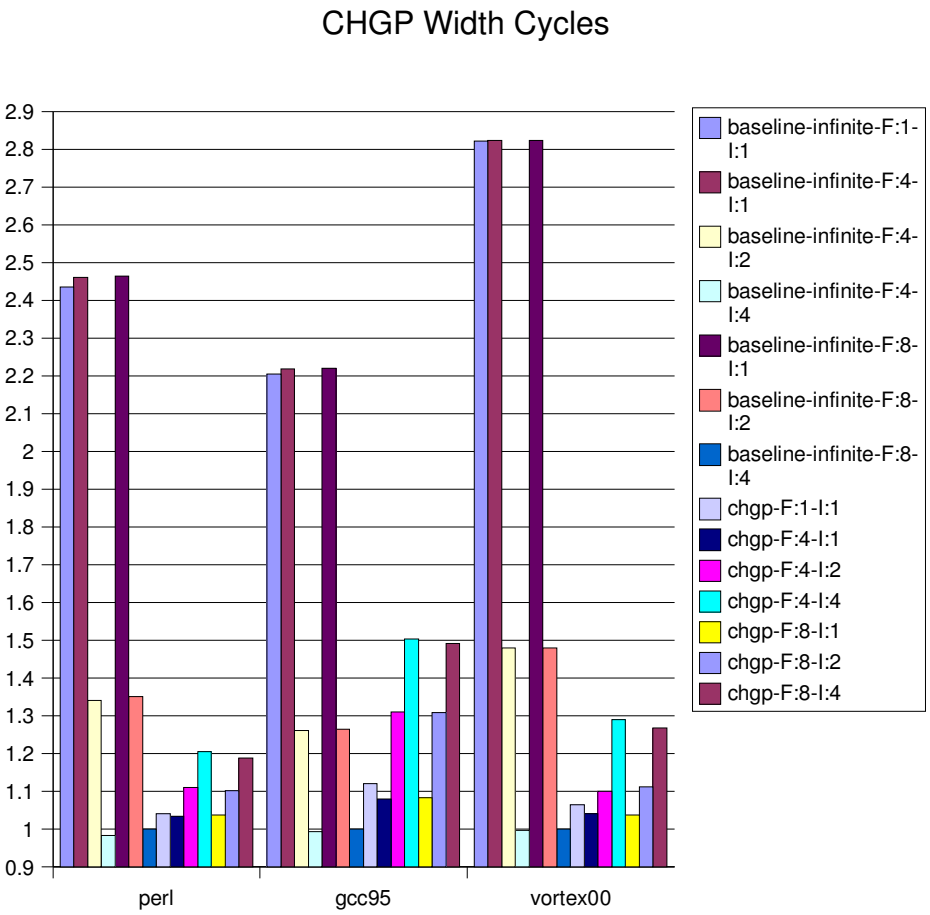


Figure 6.56: CHGP Width Performance

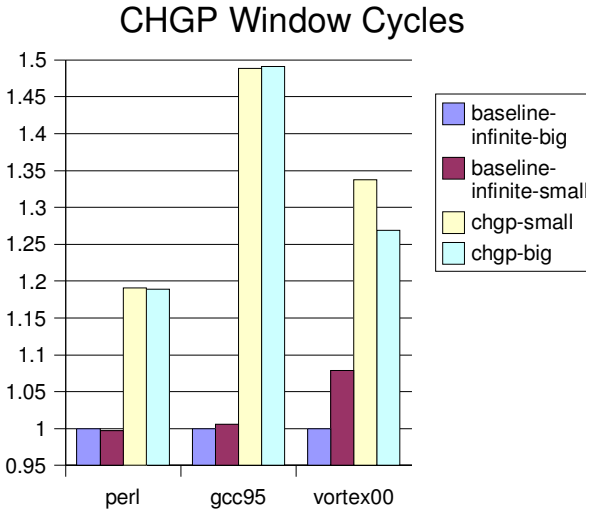


Figure 6.57: CHGP Window Performance

Architectural Impact

Figures 6.54, 6.55, 6.56 and 6.57 present the performance of CHGP under the same architectural models used for BHGP. In general the performance trends are the same as with BHGP, a not surprising result. CHGP is affected slightly more by reduced cache bandwidth and by reduced window size on the vortex benchmark.

6.6.5 Cooperative CHGP Results

We present the performance of Cooperative Compiler Hint Guided Prefetching in figure 6.58. Cooperative CHGP outperforms sequential-eight prefetching alone by up to 6%, though the gain for vortex is a modest 1%. Again we observe that operation in fetch degrades performance. In this case we speculate that Cooperative CHGP suffers the same problems as Cooperative Prefetching: prefetches are issued too early and are kicked out before they become useful.

The prefetch slack measurement for Cooperative CHGP appears in figure 6.59. 13% of the prefetches are late, indicating that Cooperative CHGP suffers in terms of timeliness compared to Cooperative Prefetching. However, 23% of the prefetches are never used, an improvement from the 39% ratio for Cooperative Prefetching.

We attribute the improved performance of Cooperative CHGP to this factor. In our implementation of the Luk and Mowry dominator prefetch optimization, we had to decide how to implement several ambiguous points in the algorithms. For example, it is not clear whether the code that moves prefetches into dominator blocks consults other filters such as checking for cache locality or coverage by the sequential prefetcher. We assumed that such filters were disabled. It is also possible that our algorithms implements these optimizations differently and results in very different prefetch scheduling.

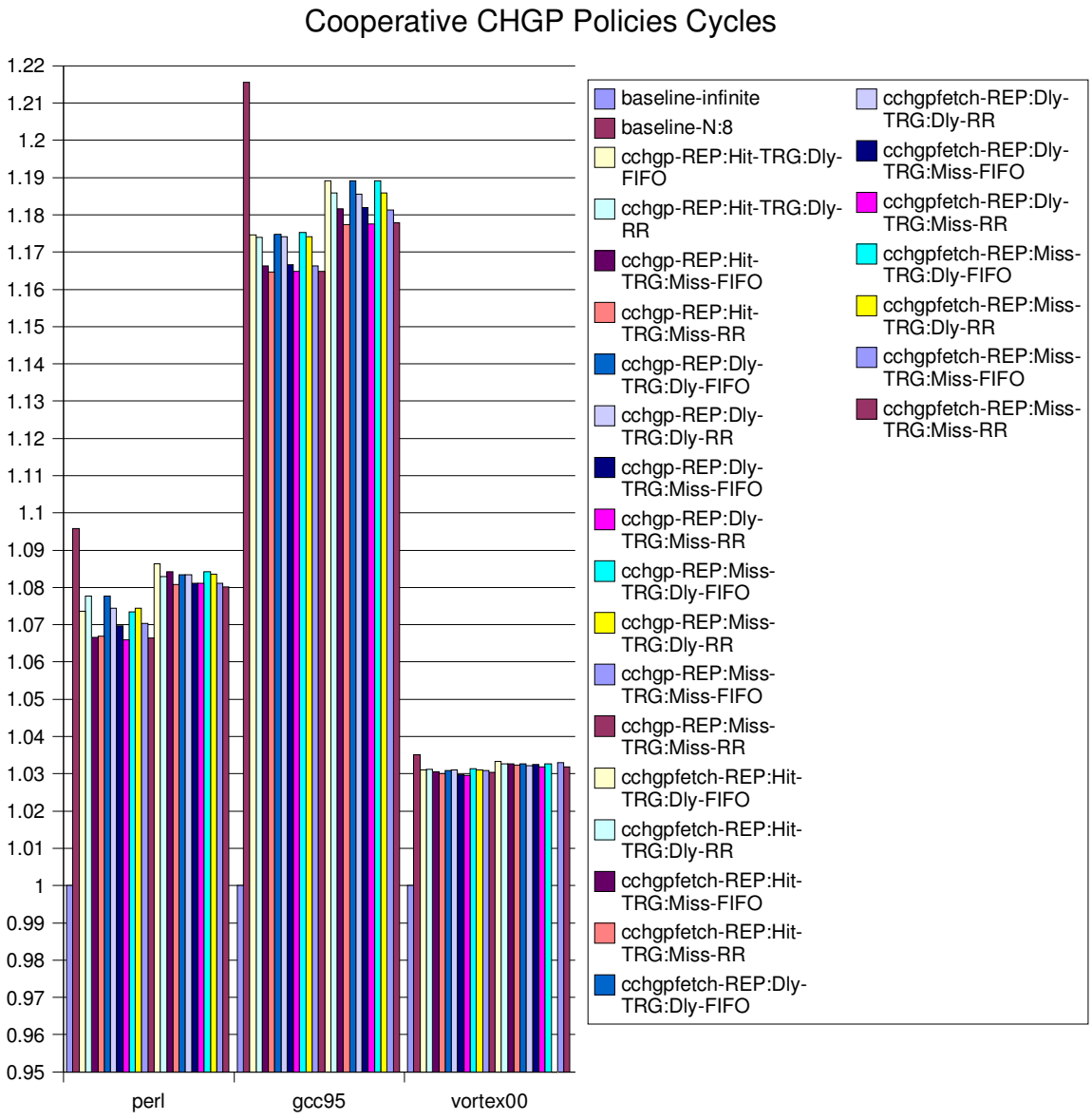


Figure 6.58: Cooperative CHGP Performance

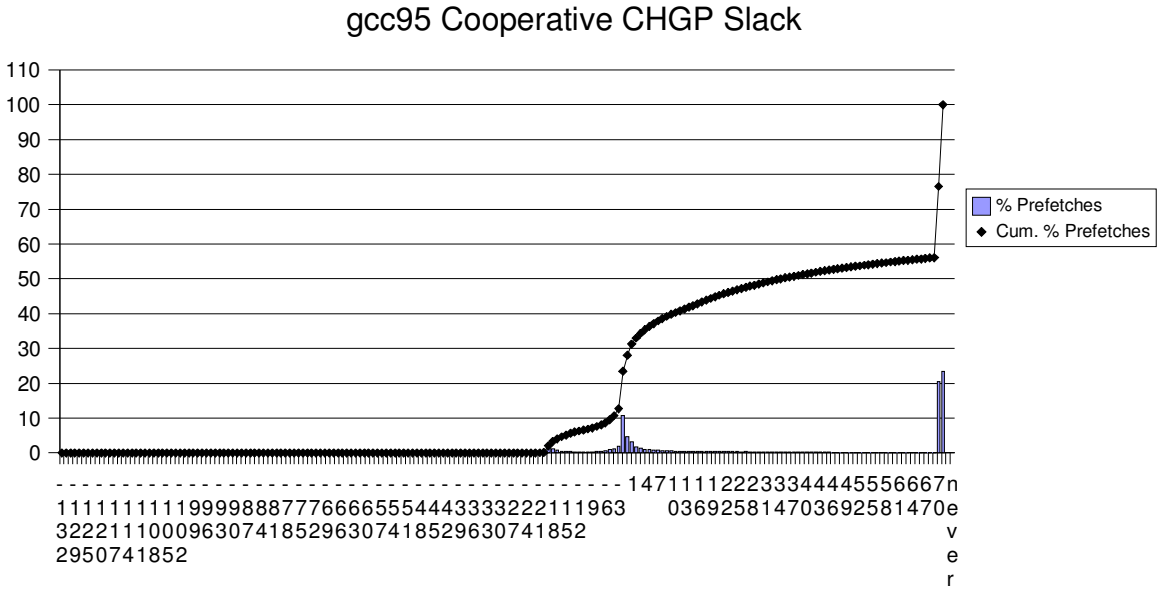


Figure 6.59: Cooperative CHGP Slack

In any case, Cooperative CHGP eliminates many of these ambiguities because it does not run the dominator optimization filters. Thus prefetches are placed more liberally than in Cooperative Prefetching and have a better chance at being useful.

Architectural Impact

Figures 6.60, 6.61, 6.62 and 6.63 present the performance of Cooperative CHGP under the same architectural models used for BHGP. In general the performance trends follow those of sequential-eight prefetching as that is the dominant source of prefetches in this scheme, as it is for Cooperative Prefetching. Cooperative CHGP suffers less for restricted cache ports than Cooperative Prefetching, 8% in the worst case vs. 13%. It also suffers less from restricted cache bandwidth. An eight-byte bus degrades performance 13% vs 22% for Cooperative Prefetching in the worst case. It also suffers slightly less from issue width effects.

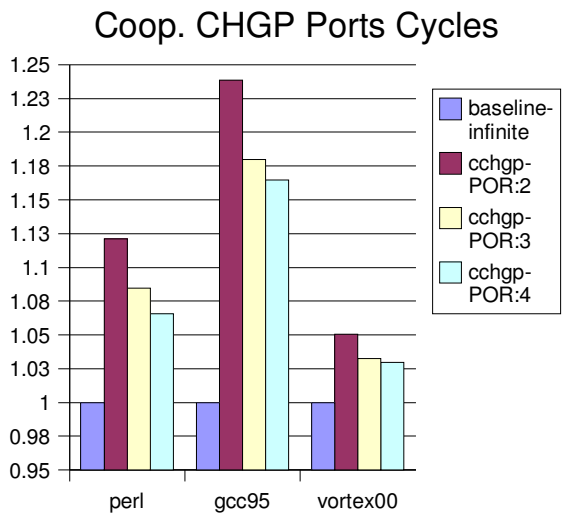


Figure 6.60: Cooperative CHGP Ports Performance

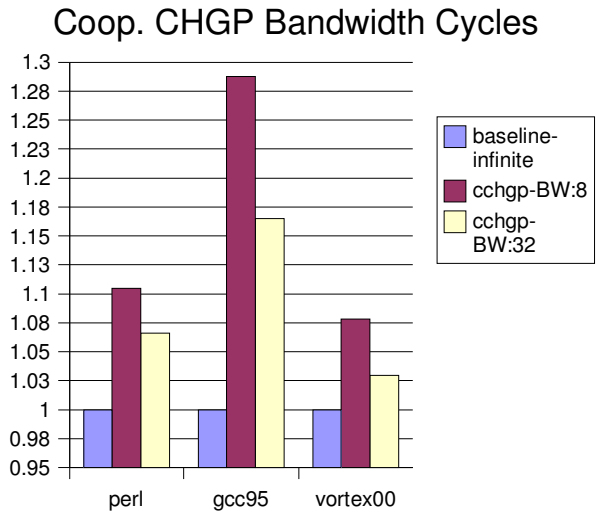


Figure 6.61: Cooperative CHGP Bandwidth Performance

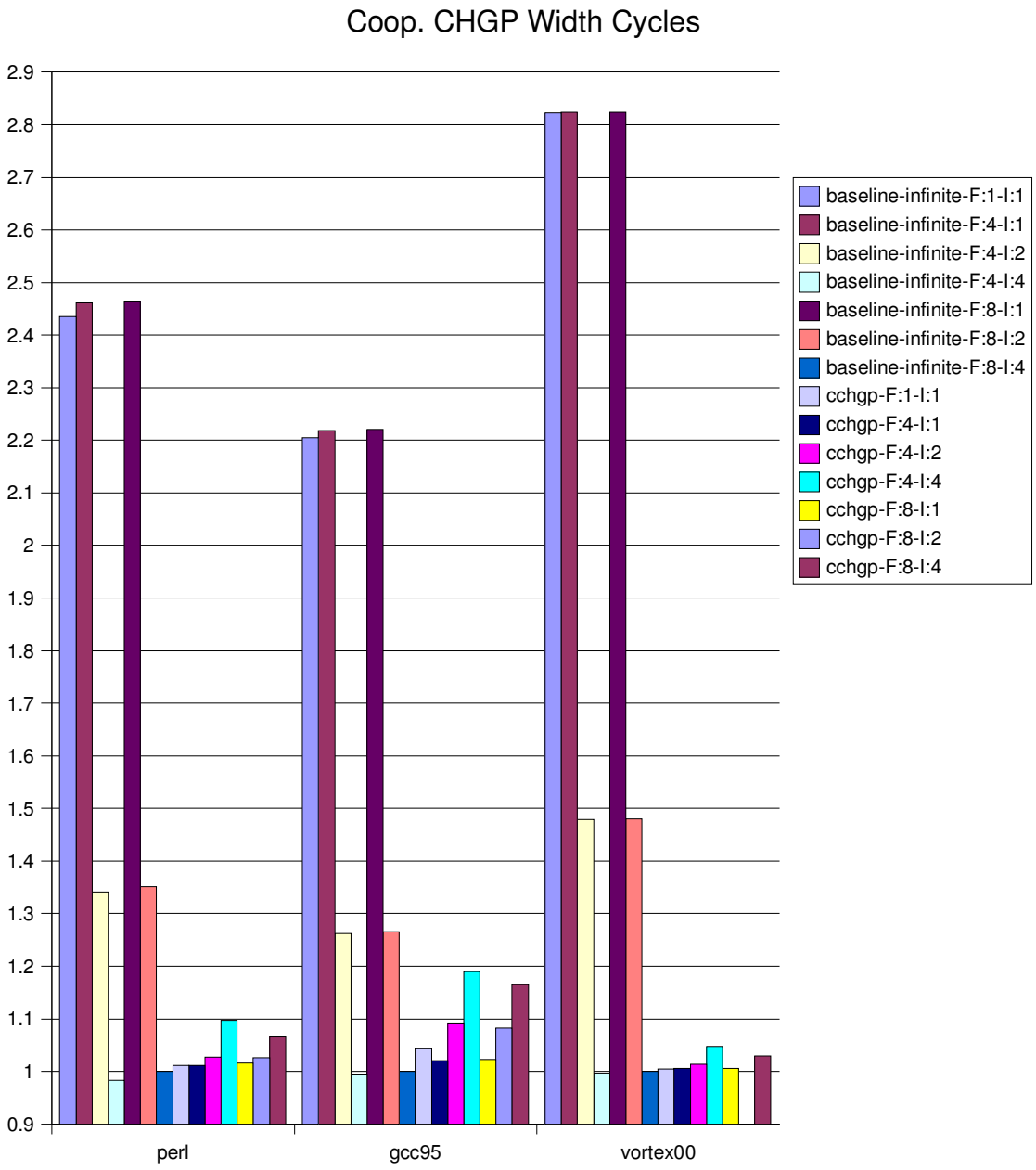


Figure 6.62: Cooperative CHGP Width Performance

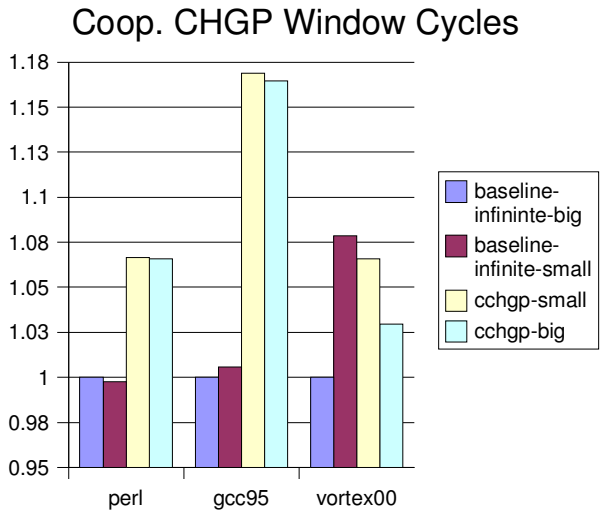


Figure 6.63: Cooperative CHGP Window Performance

Fetch-Based Cooperative Prefetching

Given our results with BHGP, CHGP and Cooperative CHGP, we wished to measure the effects of operating Cooperative Prefetching in the fetch stage of the pipeline. Figure 6.64 shows the results. The performance impact is negative but negligible. We speculate that the fetch-based prefetcher issues prefetches even earlier than the decode-based prefetcher and since the decode-based prefetcher already issues a high ratio of useless prefetches, operating it in the fetch stage only exacerbates the problem. The fact that the decode-based prefetcher already has a high rate of useless prefetches implies that there is less room to further degrade performance so the additional penalty is not significant.

6.7 Previous Work

There is a wealth of published materials covering instruction fetch efficiency. In this section we survey a subset of this work and identify points of difference between them. We show that while many studies of instruction prefetching have been performed, the baseline

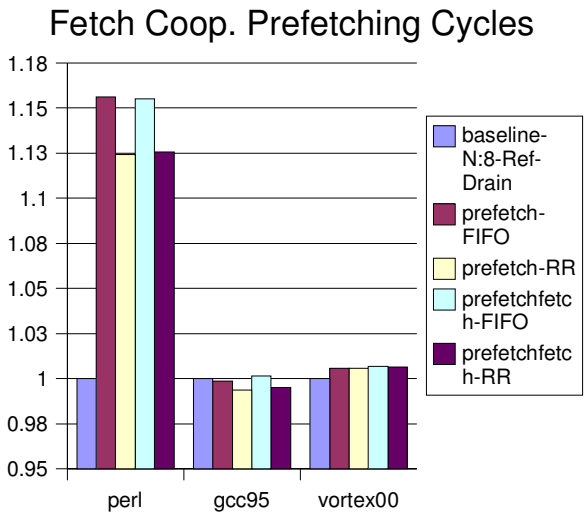


Figure 6.64: Fetch-Based Cooperative Prefetching Performance

assumptions used render them nearly incomparable. Even more distressing, a number of studies leave important assumptions unstated, making reproduction of the experiments impossible.

Smith studied instruction prefetching and cache memory organization extensively [76, 67]. Out of necessity, these studies employed trace-based simulation and did not account for contention within the memory subsystem. While various aspects of cache design such as associativity and replacement algorithm were considered, the only type of prefetching considered is sequential prefetching of distance one. Three variants are explored: prefetch on fetch, prefetch on miss and tagged prefetching, which prefetches the next line only on the first demand reference to a line. Thus tagged prefetching acts like prefetch-on-miss except that demand references that hit in the cache due to a previous prefetch also initiate a next-line prefetch. Prefetch-on-miss was determined to be the least effective method, with tagged prefetching providing the additional benefit of reduced cache bandwidth requirements. The studies in this chapter are closer to tagged prefetching because the confidence mechanism

employed by the prefetch engine tends to filter out useless prefetches. While Smith did not consider sequential prefetching distance of greater than one feasible, modern banked or multi-ported cache architectures present opportunities for greater fetch-ahead distances.

Smith and Hsu revisited sequential prefetching in the context of pipelined and super-scalar architectures [60]. Two main prefetching strategies are explored: sequential prefetching and target prefetching. It is not clear whether a prefetch-on-reference or prefetch-on-miss strategy is used for the sequential prefetcher. Target prefetching attempts to look beyond control-flow changes to prefetch non-sequential lines. A line target prediction table is used to generate a prefetch address given a demand fetch address. In both schemes only a single line is prefetched at a time.

Performance for target prefetching is roughly equivalent to the sequential prefetching strategy. Because the target prefetcher simply keeps a table of likely target addresses, it subsumes a sequential prefetcher because the table can include addresses for the next sequential line. To prevent the table from filling with sequential addresses, a combined technique that employed both strategies was examined. Performance was found to be best with this strategy.

Jouppi proposed stream buffers to implement sequential prefetching with distances greater that one [72]. Stream buffers operate in a sequential manner. We additionally modeled a sequential prefetcher than can operate in parallel, taking advantage of free cache ports to prefetch multiple lines ahead simultaneously. Stream buffers only serve their head queue entry as a potential fetch hit target. If the data is available further down the queue it will not be seen. Our model assumes the prefetch buffer can be accessed associatively. Stream buffers automatically fetch the next sequential line when an entry is removed from the buffer while our sequential prefetch model is given an explicit prefetch distance after

which it will terminate.

Pierce and Mudge explored the effects of branch misspeculation on the instruction fetch engine [68]. Their wrong-path prefetching scheme combines sequential prefetching and prefetching of branch instruction targets. Because the targets are not known until after the decode stage of the pipeline there is some delay before the non-sequential prefetch may be initiated. This scheme prefetches both targets of branches in the fetch line, the fall-through path via the sequential prefetcher and the non-sequential path via the branch target prefetch. Though not explicitly stated, it can be assumed that the sequential prefetcher only prefetched a single line at a time.

The cost of such a scheme is lower than the target and hybrid prefetching schemes of Smith and Hsu because no prediction table is needed. This study found the hybrid scheme to be only marginally better than sequential prefetching, pointing to the high degree of sensitivity of instruction prefetching to architectural assumptions. Wrong-path prefetching was found to be slightly better than sequential, target and hybrid schemes.

Xia and Torrellas explored instruction prefetch in the context of operating system codes [66]. In their scheme the compiler marks instructions with hint bits. At run-time a hardware sequential prefetcher can run ahead of the main program control until a hint bit is encountered. The hint tells the hardware prefetcher that the prefetch sequence is not likely to be productive beyond that point and thus the hardware prefetcher stops. The sequential prefetcher is activated on a cache miss . If the hardware prefetcher is in the middle of a prefetch sequence when the demand miss is encountered, the prefetcher is redirected to start a new sequence after the miss address. To recover the additional advantage of prefetching on a demand hit, an idle prefetcher will start a new sequence if the demand reference is outside of the sequence previously prefetched. This scheme forms the basis for our Ad-

vanced prefetch queue implementation. Rather than encoding stop bits into the instruction scheme, our sequential prefetcher is given a fixed-length sequence to prefetch.

In addition, Xia and Torrellas study hardware and software schemes to prefetch branch targets. In the hardware scheme, branch instructions are decoded and the target address is prefetched. In the software scheme prefetch instructions are inserted in the basic block containing a branch whose target is prefetched by the instruction. They found that prefetching such transition misses, whether in software or hardware, was detrimental because it disrupted the sequential prefetcher through cache pollution. We have observed this detrimental effect as well, even when more advanced software algorithms are used to schedule prefetches further away from branch targets. Like Xia and Torrellas, we conclude that a multi-line sequential prefetcher with some sort of confidence-based filtering mechanism outperforms even sophisticated techniques to prefetch transition misses.

Luk and Mowry explored more complex software prefetching algorithms and developed Cooperative Prefetching to cover misses over long-distance control transfers [45]. Unlike Xia and Torrellas and the studies of this chapter they report good results with such techniques. Unfortunately, as outlined in sections 6.2, 6.3 and 6.5, enough ambiguity in the software algorithms and machine model exists to make a completely fair comparison impossible. We do not discount the possibility that our algorithms and architectural models vary widely from those used by Luk and Mowry. Unfortunately, efforts to clarify such questions were unsuccessful [77].

Srinivasan, *et al.* developed the Branch History Guided Prefetching (BHGP) scheme outlined in section 6.2. Their study compared BHGP to a sequential prefetching at the mBTB prefetching technique [78]. The sequential prefetcher uses tagged prefetching to prefetch one line ahead of fetch. The mBTB technique uses a complex branch target buffer

to generate multiple prefetch candidates of branches K-1 branches past the current branch being executed. The most likely target is selected for prefetching based on the values of saturating counters.

BHGP was found to outperform both the sequential and mBTB prefetching techniques. The primary advantage over sequential prefetching was the greater latency tolerance due to issuing prefetches five branches ahead of their targets. Because the BTB of mBTB grows exponentially with lookahead length, K was limited to small values and thus suffered many of the same timeliness problems.

While the sequential prefetcher modeled employed tagged prefetching to limit useless prefetches, it is not clear whether the mBTB technique, as modeled, used any filtering mechanism. BHGP employed a one-bit confirmation counter in the L2 tags, similar to the 2-bit counter used by Cooperative Prefetching and in the studies of this chapter.

Call Graph Prefetching was proposed by Annavaram, *et al.* in hardware and software variants [63, 64]. The hardware variant uses a Call Graph History Cache to provide the addresses of procedures called by a particular routine. Each callee is prefetched in sequence after the previous callee returns. Sequential prefetching is used to prefetch within a procedure. This variant performed about 7%-10% better than sequential prefetching alone. It is not clear whether the sequential prefetcher could prefetch multiple lines or only a single target. The software variant is outlined in section 6.2. Speedups of up to 24% were observed compared to no prefetching. No comparisons to other prefetching techniques were reported and the machine model presented only listed simple memory subsystem parameters, excluding important characteristics such as instruction window size and issue width.

Reinman, Calder and Austin studied instruction prefetching in the context of a scalable fetch engine [71]. The fetch architecture decouples the instruction cache from the branch

predictor by introducing a new structure called the *Fetch Target Queue* (FTQ) [79]. Their *fetch-directed instruction prefetching* (FDP) scheme marks fetch targets in the FTQ that are predicted to be good candidates for prefetching. As in our machine model, they include a Prefetch Instruction Queue (PIQ) to hold prefetch requests and a small FIFO buffer to cache prefetched blocks Various strategies to determine which entries in the FTQ should be prefetched. FDP was found to out-perform sequential prefetching and streaming buffers but combination techniques utilizing FDP and sequential prefetching or streaming buffers was found to work well for low-bandwidth and high-bandwidth cache memory architectures, respectively.

The interplay between the branch predictor, instruction window and cache was explored by Skadron, *et al.* [75]. This work emphasizes the importance of accurate machine modeling, in particular of the cache and branch predictor. Their finite instruction cache penalties are more in-line with our results here than those of Luk and Mowry in the Cooperative Prefetching study. Furthermore, they found that imperfect branch prediction and finite, small instruction caches limit the benefits of a large window, as we verified in section 6.6.1.

The trace cache is a new architecture for instruction caches proposed by several researchers [80, 81]. The trace cache attempts to pack dynamically sequential instructions together into a *trace*, essentially folding away any taken branches that led to the particular trace being built. This is a hardware version of the static trace scheduling developed by Fisher [31]. By removing dynamic branches, the trace scheduler and trace cache can reduce the number of non-sequential instruction cache misses and improve the effectiveness of simple prefetching techniques. The trace cache has the additional benefit of potentially packing more useful instructions into a cache block by replacing useless instructions after a taken dynamic branch with the instructions at the branch target.

The dynamo project explored dynamic software instruction translation and optimization [82]. Part of the dynamic optimization process includes elimination of dynamic branches by reordering code according to the dynamic paths encountered. In this way the optimizer acts as a sort of software trace cache, achieving some of the benefits of the hardware trace cache variants.

6.8 Conclusion

In this section we summarize the results of our experiments and make suggestions for prefetch designs based on the studies in section 6.6. In addition, we propose some ideas for future study based on our experiences with instruction prefetching.

6.8.1 Summary

Overall, sequential prefetching is a highly effective technique, especially when operating over large distances as with the eight-line prefetcher in our study. Most transition prefetchers such as BHGP and Cooperative Prefetching cannot keep up with aggressive sequential prefetching. Cooperative CHGP shows some promise though it suffers the same table size problems as BHGP.

It is possible that larger programs may paint a very different picture. Larger programs require larger caches and this may reduce the relative table size of BHGP and related prefetchers. However, we caution that larger programs also produce larger miss streams and the table size may have to increase to compensate.

Even so, sequential prefetching leaves much room for improvement. With an 8K cache, gcc95 still sees a 22% performance degradation with eight-line sequential prefetching over an infinite cache machine. Some of this is recovered by Cooperative CHGP, which reduces

the penalty to 16%. Still, there is a large gap in performance yet to be recovered.

6.8.2 Future Work

Our experience with the software prefetchers has not been a happy one. We have found the compiler heuristics for statically modeling cache behavior to be inadequate. The bloat caused by instruction prefetch instructions often outweighs the benefit of the additional prefetching. Luk and Mowry do report better results when profiling is used to determine the program miss behavior, but such profiling is not always an option.

Due to these problems we studied the CHGP and Cooperative CHGP schemes. An additional option is to examine schemes that insert prefetch instructions but do so more conservatively than in Cooperative Prefetching. Annavaram's software variant of Call Graph Prefetching provides an inspiration for another potential design: Cooperative CGP. This scheme, like Cooperative CHGP relies on software prefetches to capture the long-distance behavior. Because no prefetch table is available, instruction prefetches for the next likely procedure or procedures to be called may be inserted after each function call instruction in the program. Some of the cache and dominator heuristics of Cooperative Prefetching may be used to reduce the prefetch bloat. This scheme may overcome the bloat problems of Cooperative Prefetching because prefetches may only be scheduled after function calls, meaning that there should be fewer prefetches scattered throughout the program text. A hardware sequential prefetch can be used to cover the space between function calls, in contrast to the even-spaced instruction prefetch instructions inserted by Annavaram's software CGP variant.

Another path of exploration involves the prefetch optimization heuristics used in Cooperative Prefetching. As we noted in section 6.6, it appears as though our implementation

of Cooperative Prefetching is scheduling prefetches too far away from their targets. By manipulating various parameters to the filters it may be possible to obtain better prefetch schedules.

Much of the prefetching design space still remains to be explored. By necessity we have only presented a sampling of potential designs. In particular, it may be interesting to explore inter-procedural scheduling algorithms to place prefetches along deep call chains to prefetch into caller routines, or vice versa. This may eliminate some of the timeliness problems of Cooperative Prefetching be allowing more flexibility in prefetch placement. Finally, large codes such a corporate database systems and productivity software may behave quite different from the small SPEC benchmarks studied here.

CHAPTER 7

Speculative Register Promotion on Modern Microarchitectures

7.1 Introduction

This chapter concentrates on the use of speculative register promotion to overcome the difficulties presented by separate compilation and side-effects. Chapter 8 presents novel extensions to handle the problem of aliasing. This chapter presents a study of speculative register promotion on modern pipelined processor microarchitectures consisting of dynamically scheduled out-of-order execution cores with register renaming. Such architectures present unique challenges for speculative register promotion. We examine these difficulties, present solutions for them and evaluate the performance of speculative register promotion on such architectures using a highly accurate simulation model.

7.2 Speculative Register Promotion

In this section we review the register promotion compiler transformation in its non-speculative and speculative forms. The latter is the focus of this study.

Register promotion is a transformation that attempts to enregister data that normally cannot live outside of addressable memory. There are several possible reasons that a particular data item cannot be placed into a register:

- It has its address taken
- It is visible outside of local scope (e.g. it is a global variable)
- It is part of a larger aggregate item
- It is not a concretely nameable location (i.e. free-store allocations)

Examples of pieces of data which traditionally cannot be placed into registers for their entire lifetime includes global variables, aliased data, anonymous memory (the results of a `malloc` call in C), array elements and structure data fields. Side-effects are the most common problem. Global variables cannot usually be placed into a register for their entire lifetime because other functions must be able to access and manipulate them. With separate compilation it is not easily possible to generate correct code such that all references to the global data are through the same register name. Parameters passed by reference to functions result in similar complications. The aliasing problem can also be common. The data pointed to cannot easily be placed into a register because another access (either via a different pointer or a direct variable reference) may alias it. The compiler cannot know in general whether two names refer to the same data if one of them is a pointer of statically unknown value and the other is a pointer with similar unknowns or a data item whose address has been potentially assigned to the pointer. The next class of complications involves aggregate data. It is difficult in general to assign, for example, C `struct` fields to registers because structure data objects are often manipulated as a whole via block copying and parameter passing.

```
int a[5] = { 0, 1, 2, 3, 4 };
int c[5];

int main(void)
{
  int i, j;

  for (i = 0; i < 5; i++) {
    c[i] = 0;
    for (j = 0; j < 5; j++) {
      /* c[i] can be removed entirely from the loop and replaced with
         a register. Register promotion is needed for this. */
      c[i] += a[j];
    }
  }

  return 0;
}
```

Figure 7.1: Register Promotion Example

Register promotion attempts to identify cases where some of these restrictions may be relaxed. An example appears in figure 7.1. Because the index to global array `c` is invariant in the inner loop, all memory references to that element may be moved out into the outer loop. Loop-invariant code motion cannot perform this transformation in general because of the inductive assignment to `c[i]`. The value `c[i]` itself is not invariant and must be processed within the inner loop. Furthermore, because there are no function calls within the inner loop, side-effects are not a concern. Array `c` will not be accessed anywhere else while program execution is still within the inner loop.

Figure 7.2 shows a high-level view of the register promotion transformation. A new compiler-generated variable `__c_promote_temp` has been generated[1]. An assignment of `c[i]` to this temporary is performed just before the inner loop. Within the loop we have replaced `c[i]` with the temporary. Upon exiting the loop the program stores the final temporary value back into `c[i]`. Because the temporary is compiler-generated it is guaranteed not to

[1]We use the reserved double-underscore name prefix to emphasize that this is an automatically-generated storage location.

```
int a[5] = { 0, 1, 2, 3, 4 };
int c[5];

int main(void)
{
  int i, j;
  int __c_promote_temp;

  for (i = 0; i < 5; i++) {
    c[i] = 0;
    /* Promotion */
    __c_promote_temp = c[i];
    for (j = 0; j < 5; j++) {
      /* c[i] can be removed entirely from the loop and replaced with
         a register. Register promotion is needed for this. */
      __c_promote_temp += a[j];
    }
    /* Demotion */
    c[i] = __c_promote_temp;
  }

  return 0;
}
```

Figure 7.2: Register Promotion Performed

be aliased anywhere and thus is a candidate for register allocation by the code generator. Register promotion has allowed the compiler to allocate global data to a register for a short lifespan, arguably the most important lifespan because it is over the body of an inner loop.

Such transformations may be applied in many different situations. The key to the success of register promotion is the lack of side-effects and aliasing over the restricted lifespan of the promoted value. Figure 7.3 throws a monkey wrench into a simplified version of the register promotion example. A function call has been added into the inner loop. As the `do_something_unrelated_to_c` name implies, we assume that the routine invoked does not touch `c` in any way. Even so, if this routine appears in another compilation unit the compiler cannot know its global data interface and thus cannot guarantee that copying `c` to a register is safe. Register promotion cannot perform any transformations in this case.

While the MIRV linker can be used to allow promotion by performing inter-procedural

```
int c;

int main(void)
{
  int i, j;

  for (i = 0; i < 5; i++) {
    c += i;
    do_something_unrelated_to_c();
  }

  return 0;
}
```

Figure 7.3: Register Promotion Failure

MOD/REF analysis, it is not always practical or desirable to run these more expensive analyses. Other compilation tool-sets may not have the ability to perform such analyses so it is desirable to find a method of performing the promotion of `c` in a separate compilation environment. Speculative Register Promotion provides the means.

As the name implies, speculative register promotion performs the register promotion transformation speculatively. That is, it performs the transformation knowing that it may be incorrect. The compiler relies on special hardware (described in section 7.4) to detect unsafe transformations at runtime and perform the necessary actions to resolve the problems. If `do_something_unrelated_to_c` actually does manipulate `c` the hardware must detect this and redirect the operations to the register holding the promoted data.

Figure 7.4 shows the results of speculative register promotion at the assembly code level. The function linkage instructions have been removed for clarity. Speculative register promotion uses two new instructions: `map` and `unmap`. These are special load and store (respectively) instructions that update the speculative register promotion hardware in addition to performing their regular data movement tasks.

```
main:
        # ...prologue
$L50:
        move $22,$0
        mapw $23,c      # Speculative promotion
$L52:
        addu $23,$23,$22 # Use and define promoted value
        jal do_something_unrelated_to_c
        addu $22,$22,1
        li $2,5
        slt $2,$22,$2
        bne $2,$0,$L52
$L53:
        unmapw $23,c     # Speculative demotion
        move $2,$0       # Return value
$L51:
        # ...epilogue
```

Figure 7.4: Speculative Register Promotion

7.3 SLAT Compiler Impact

Speculative register promotion has a number of consequences for the compiler. In addition to the register allocator modifications necessary to implement the promotion, the compiler must make sure not to violate program semantics. When a data object is speculatively allocated to a register, the transformation filters must be aware of the speculation. Instruction schedulers, for example, cannot move store operations above a speculatively enregistered load unless the addresses can be guaranteed not to conflict. Because the enregistered load looks like a register copy to the compiler, state must be maintained to flag the copy as speculative.

Similar care must be taken in other transformation filters. Copy propagation, for example, cannot move a speculatively enregistered value past a memory operation unless the addresses can be disambiguated statically. Many transformations involve some type of code motion and each must take the same care to avoid violating memory ordering.

If a speculatively enregistered value must be spilled due to register pressure, the compiler

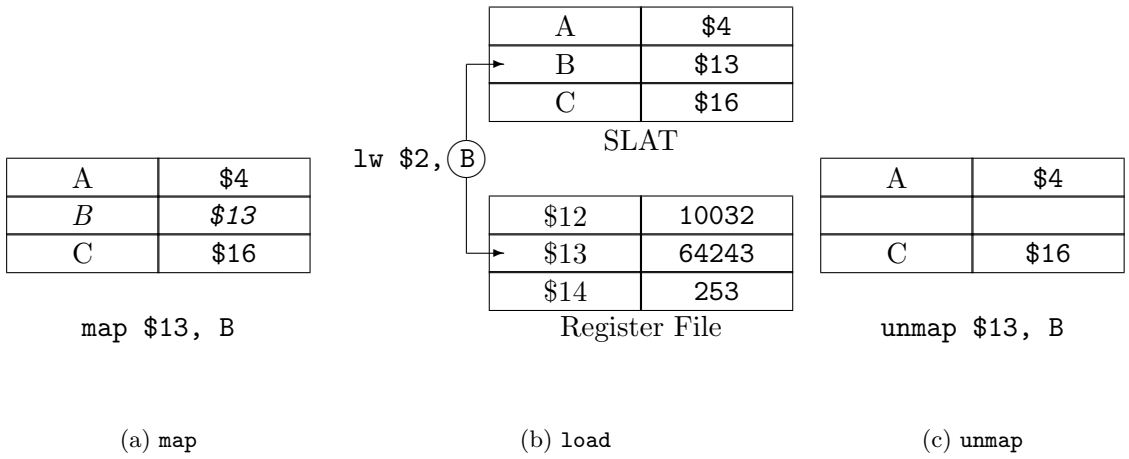
must take care to undo the speculation. This is because spill instructions have special semantics tied to the machine architecture that supports speculative register promotion (c.f. sections 7.4 and 7.6). Such instructions are more expensive than "regular" register spills and reloads so reversing the speculation will improve performance of the code. Furthermore, the prototype ABI described in section 7.6 keeps a machine bit-vector word to track which registers are relevant to the hardware structures. Spurious register pressure spills and reloads can upset this information and result in incorrect program execution.

7.4 The Store Load Address Table

As mentioned in the previous section, speculative register promotion requires hardware support to detect and correct problematic transformations performed by the compiler. In previous work we have proposed the Store-Load Address Table (SLAT) to provide this mechanism [49].

The SLAT is an extension of the Advanced Load Address Table (ALAT) introduced by the Intel Architecture 64 (IA64) [38]. The ALAT provides hardware support for static speculative load scheduling. At compile-time the code generator may schedule a load before a potentially conflicting store instruction. A check instruction is placed at the original load location. At run-time the advanced load reads the specified address and enters that address into the ALAT. If a later store accesses that same address the ALAT entry is marked. When the check instruction is executed the ALAT entry for the specified address is queried and if the entry shows a store conflict program control branches to compiler-generated fixup code, which must re-load the correct data. If the ALAT becomes saturated entries are dropped and their corresponding checks will fail, triggering a (potentially unnecessary) fixup.

The SLAT uses the ALAT concept of mark-and-check for an entirely different pur-



(a) `map` (b) `load` (c) `unmap`

Figure 7.5: SLAT Operation

pose. The SLAT allows speculative register allocation. While the speculative scheduling performed by IA64 compilers only affects the single load instruction scheduled, register allocation affects both load and store instructions in multiple contexts. Because some store instructions will be converted to register writes, this implies that load instructions must query the SLAT to see if the data actually resides in a register. The name "Store-Load Address Table" derives from the fact that both load and store instructions must query the hardware structure.

Operation of the SLAT is straightforward. A `map` instruction is a special type of load that associates an address with a logical register tag in the SLAT, as shown in figure 7.5(a). All memory instructions must query the SLAT for their source (load) or destination (store) address. If there is a match the memory operation is then redirected to the register specified in the SLAT entry as shown in figure 7.5(b). We refer to this memory operation as *conflicting* because it accesses data that was placed into a register by speculative register promotion. In other words, the transformation was unsafe. The `unmap` instruction is a special store

```
int main(void)
{
  map r5, global
  for(...) {
    r5 = r5 + 1
    foo()
  }
  unmap global, r5
  return(0);
}

void foo(void)
{
   map r6, global
   for(...) {
      r6 = r6 + 1
      bar()
   }
   unmap global, r6
}
```

Figure 7.6: Multiple Register Mappings

that removes the specified address from the SLAT, dissociating it from the logical register.

The above operation assumes that only one register may be "active" for a given address at any time[2]. It is possible for a single address to be *mapped* to two registers at a time but only one of them will be considered active. This is explained by the code in figure 7.6. In function `main` global variable `global` is mapped to register `r5`. When function `foo` is called, it maps `global` to `r6`. This `map` instruction will conflict in the SLAT an be converted to a copy from `r5` to `r6`. Clearly at this point `r6` holds the current value of `global` and all references to `global` should be directed to `r6`. This can be accomplished in a number of ways. A simple scheme keeps a counter in each SLAT entry that identifies the dynamic runtime stack frame within which each mapping was performed. SLAT lookup then simply chooses the entry indicating the most recent frame. We assume a scheme such as this throughout our studies.

[2]But see chapter 8 for more powerful SLAT configurations.

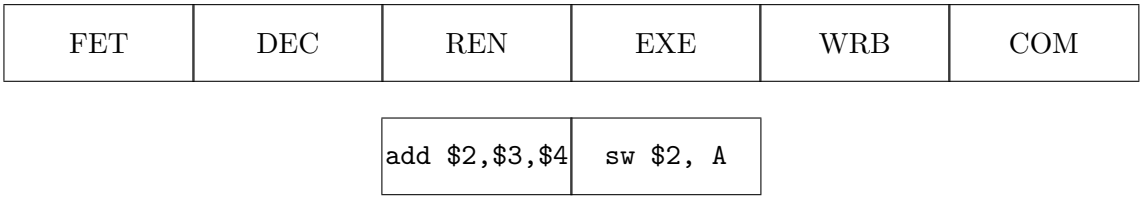
Because the SLAT holds important mapping information about how memory addresses relate to registers it cannot drop entries as the ALAT does. If the ALAT drops an entry it simply forces the check instruction to branch to fixup code which will re-load the correct data at some point. Speculative register promotion potentially affects many different instructions, converting memory operands to register operands. Once that conversion has been performed there is no way to convert them back to memory operations if a SLAT entry is dropped. The key difference between speculative load scheduling and speculative register promotion is that a speculative load retains its original semantics while being given additional operations to the instruction (entry into the ALAT) while speculative register promotion completely changes the semantics of many data references because a logically independent memory space is substituted for the original data location.

Fortunately, the SLAT need only hold as many "active" entries as there are logical registers. Since the compiler cannot allocate any more registers than are available in the machine architecture, it cannot create any more "active" mappings than the number of such registers. Just as with the registers themselves, SLAT entries can be saved to the runtime stack and restored when needed.

The experiments in this dissertation assume an infinite-sized SLAT to gauge the potential improvement provided by speculative register promotion. Therefore, our compiler and simulator do not measure the effects of SLAT entry save and restore. A production system would require compiler and hardware action to save and restore SLAT entries as needed. A prototype Application Binary Interface (ABI) for the SLAT appears in section 7.6. Alternatively, a mechanism such as IA64's register save engine could be adapted for the SLAT to provide a logically larger table [38].

A	$4
B	$13
C	$16

(a) SLAT Contents



(b) Pipeline

Figure 7.7: SLAT Register Renaming Impact

7.5 SLAT Architectural Impact

Our previous work and Postiff's dissertation studied speculative register promotion with the SLAT in a fairly abstract manner [49, 4]. These studies, like ours, assumed an infinite-sized SLAT. The studies also primarily focused on instruction count and the number of memory operations required to execute the benchmarks. These numbers were obtained by executing the benchmarks on a purely functional simulator. While these numbers provide some indication of performance improvement, we desire a more concrete measurement. The experiments in section 7.7 are executed on a cycle-accurate simulator for a modern out-of-order microprocessor.

Implementation of the SLAT on an out-of-order microarchitecture reveals various complications not addressed by previous work. Because the SLAT can dynamically change the semantics of a memory instruction from a core memory access to a register reference, the microarchitecture must be capable of recognizing and possibly recovering from this situation. The primary difficulty is in the register renaming stage of the machine pipeline.

Figure 7.7 shows a simple six-stage pipeline and the contents of the SLAT at some point in time. Because the address referenced by a load or store instruction is not available until just before the instruction enters the execute stage, the microarchitecture cannot properly track data dependencies through the register to which the address is mapped in the SLAT. Because the SLAT holds logical register tags, the microarchitecture must rename it before performing the read or write operation. Unfortunately, renaming in the execute stage is too late because dependent instructions (converted from memory operations by speculative register promotion) may have already passed the register rename stage in the time between rename of the memory instruction and execute of that instruction in the processor core. They will not have picked up the proper tag from a converted store instruction. In figure 7.7 the store will be converted to the register copy `move $4, $2` and the add instruction will pick up the wrong tag for $4. For load instructions the problem runs the other way. By the time the load is able to rename its mapped register, later instructions that write to that register may have already passed through register rename, resulting in an incorrect tag assignment to the load source register since the load should have renamed its (SLAT-mapped) logical source register to the older physical tag.

One obvious possible solution to this problem is to store physical register tags in the SLAT rather than logical register tags. Unfortunately, this does not truly solve the problem, but only shifts it to another part of the machine. If the SLAT stores physical register tags, then those tags must be updated every time the corresponding logical register is renamed to a new physical tag. This seriously complicates the pipeline control, likely extending the register renaming stage to multiple cycles. Furthermore, we still have the same renaming problem for memory instructions. A store will still need to rename the register in the execute stage because it is still writing a new value to that register. Later instructions that

source that tag will still pick up the wrong tag and not see the value written by the store instruction. A load will still pick up the incorrect physical register tag from the SLAT if later instructions have already renamed it in the rename stage of the pipeline.

We refer to the above renaming problem as *out-of-order rename* because memory instructions that hit in the SLAT must rename their register operands out-of-order with respect to the rename stage. In effect stores must "back patch" instructions that have referenced the corresponding logical register in the time between the store dispatch and execute. Loads must "remember" the proper tag for the register to which is address is mapped.

Because loads do not change the renaming state of their conflicting source memory operands it is somewhat easier to solve that part of the problem. We explore two different solutions to the load out-of-order rename problem. Both solutions literally do what is suggested at the end of the previous paragraph: "remember" the proper physical register tags from when the load passed through register rename. Because the load cannot query the SLAT until it executes we have no idea which register tag it will need. The solutions diverge in their approaches to this sub-problem.

Our first solution simply carries a snapshot the entire rename table down the pipeline with the load. For machines with many registers this is obviously problematic because of the amount of state present in the renaming engine. Since one of our goals is to motivate the utility of machines with large register files this is a fairly unattractive solution. There is another complication with this approach. Because we assume that the load will not conflict in the SLAT until proven otherwise, it will remain in the instruction queue as a load. Any later writes to the register with which the load conflicts may execute before the load, meaning that when the load conflicts in the SLAT and accesses the source register,

it will obtain an incorrect value. To solve this problem we would need to carry the entire register state along with the load as well and that is clearly impractical. An alternative solution would prevent register writes from executing ahead of an earlier load but this is also clearly not desirable.

The second solution simply predicts the logical register to which the load will map. Studies have shown that communication through memory is rather stable in that stores and loads can often be grouped into producer-consumer pairs [83, 84]. Because the SLAT is responsible for detecting such communication and redirecting it to registers, it is reasonable to assume that fairly good prediction rates of SLAT-mapped registers can be obtained. Such a predictor must actually make two predictions: one to decide whether the load is mapped in the SLAT and then to decide which register may be mapped.

The downside to this solution, of course, is that mispredictions will occur and the machine must recover for them. Fortunately, a misprediction of an address presence in the SLAT and a misprediction of the actual register mapped can be handled in the same fashion. If such a misprediction occurs, the load with either have not been given a physical register tag or it will have been given the wrong one (corresponding to a different logical register). Either way the load will receive the wrong data in the execute stage. If a load is incorrectly predicted to conflict in the SLAT and it does not, then it may have executed out-of-order with respect to a dependent store because it was converted to a register copy and this situation also requires recovery.

Recovery can be handled in many different ways. The effects seen by the microarchitecture are equivalent to those of branch mispredictions and most of the recovery schemes used there can be employed for SLAT mispredictions. The pipeline must be flushed and fetch re-started from the correct instruction. In this case the correct instruction is the memory

operation that was mispredicted. There is an additional twist, however. The memory operation must remember which register it actually mapped to. This can be done either by saving the state off and re-executing the instruction as a register copy or by updating the predictor before re-fetching the instruction.

Even though logically all branch misprediction recovery schemes should work, as a matter of practicality some are better than others. Strategies that checkpoint machine state such as the register rename table become very expensive because every memory operation will require a checkpoint of the state. It may be more appropriate to choose strategies used by current load and store speculation schemes as they have been tuned to perform efficiently in the presence of many memory operations. In our simulations we assume an abstract recovery mechanism that operates within the same number of cycles as branch recovery. This is optimistic compared to some of the memory misspeculation recovery schemes but will help put an upper bound on SLAT performance.

As stated earlier, conflicting store instructions are a bit more complicated than loads because they change register rename state. Fortunately, we already have the necessary hardware to handle them. A store can also query the SLAT predictor to obtain a logical register to which it may be mapped. Using this logical register, the store can query the renamer for a new physical tag. Later instructions that source that register will pick up the new physical tag as they should. As in the load case, a misprediction requires a pipeline flush and re-execute of the store. Unlike the load, all cases of misprediction require recovery. If a store is predicted to hit in the SLAT and it does not, it has incorrectly updated rename state. To save the recovery penalty the machine could store the data into the new physical register in addition to writing it out to memory. In this case the only impact will be from dynamic instructions incorrectly waiting on the store to complete before they may be

scheduled for execution. Correct machine state is still maintained.

It is important to note that `spill` and `reload` instructions that operate on registers in the SLAT do not require recovery. This is because their register operand names are known in decode and the SLAT can be queried before the rename stage is encountered. These instructions will of course check their memory address operands when they execute but unless the programmer or compiler has done something very strange they should never be found in the SLAT unless the machine is on a misspeculated path. This is because these addresses are compiler-generated stack locations and there is no way the programmer can reference them save through assembly-language trickery.

7.6 A Prototype SLAT ABI

As mentioned in section 7.4 the SLAT requires compiler support to save and restore entries when it becomes full. This section proposes a prototype ABI for the SLAT to accomplish this task. The ABI describes how SLAT entries are saved and restored, their formats and other necessary information to accomplish the necessary tasks.

Because the SLAT bridges the gap between static speculative optimization and full dynamic information, hardware support is necessary to fully implement the ABI. In some sense the compiler must "speculatively" assume any register could be mapped in the SLAT and take appropriate action.

7.6.1 Register Save/Restore Enhancements

The current PISA ABI, based on the MIPS UNIX System V ABI, specifies a set of callee- and caller-save registers [52]. At the top of a function, all callee-save registers used within the body of the function must be saved to the stack. The same set of registers must

be restored upon exit from the function.

Because the SLAT can map memory locations to registers, special action must be taken when saving or restoring such registers. In our original work we described in general what must happen: if a saved register is mapped in the SLAT, its data must be written out to the *home location*, the address to which the register is mapped in the SLAT [49]. This guarantees that any references to that address will find the correct data. Because the instruction that saves the register specifies an address on the runtime stack, that location can be used to save the SLAT entry itself. That is, the address to which the register is mapped will be saved onto the runtime stack. When the register is restored, the operations are reversed: the address is loaded from the runtime stack and re-mapped in the SLAT to the register specified in the restore instruction. The data itself must be re-loaded from the home location.

This simple description is logically correct but lacks some details necessary to make it work in a production machine. The most obvious complication is at the restore site. Because the compiler cannot know whether the register will be mapped at runtime, there is no way it can specify that the reload operation should update the SLAT and redirect the data load to the home location. A runtime mechanism is required to keep track of this.

We propose a simple bit-vector scheme to overcome this difficulty. Each bit represents one logical register, indexed by the register number. Whenever a register is mapped in the SLAT with a `map` instruction, the corresponding bit will be set. When it is unmapped with an `unmap` instruction the corresponding bit is reset. Upon entry to a function, all instructions that write callee-save registers to the stack will examine the bit corresponding to their source register. If the bit is set then data will be written to the home location and the mapped address will be saved to the stack location specified. Once all callee-save

registers have been saved, the bit-vector itself will be saved via explicit compiler-inserted instructions. The bit-vector will then be cleared before the function body proper begins execution. The function epilogue will perform the steps in reverse, reloading the bit-vector before restoring callee-save registers. Each restore instruction will query the (re-loaded) bit-vector and if the bit corresponding to its source register is set, the address will be read from the stack, mapped in the SLAT and the data reloaded from the home location.

As mentioned in section 7.3, register-pressure spills and reloads can upset this bit-vector information. This is because the spill and reload will modify the bit-vector outside of the function prologue or epilogue. After the spill, the reload will not see the bit corresponding to the spilled register set and will not know that it should be re-mapped in the SLAT. We cannot use a special register-pressure spill instruction to avoid clearing the bit because the register may be used for some other data[3] which is not mapped in the SLAT. Further spill operations on the register will incorrectly manipulate the contents of the SLAT. Due to these complications, any speculative register promotion that is spilled due to register pressure must be reversed completely.

The above scheme is not without its complications and drawbacks. A machine with many registers (256 in our experiments) will have a rather large bit-vector to save. On the 32-bit PISA architecture it would take 8 additional prologue and epilogue instructions to save and restore all of the bit-vector information for the integer registers. The equivalent overhead would be necessary to handle floating pointer registers. Fortunately, few functions require that many registers even with aggressive optimizations and speculation [4, 85]. We can take advantage of this fact to reduce the bit-vector overhead. Since we will only use a subset of the full register set for any one function, we need only save the bits corresponding

[3]That is why it was spilled in the first place!

to the registers we actually use. It is convenient to save 32 bits at a time on the PISA architecture and it is straightforward to alter the register allocator to allocate registers that are within the same 32 bit bit-vector word. A function that requires more than 32 registers will need to save two bit-vector words. This change requires that we only clear those bit-vector words we have saved. This operation can be easily incorporated into the instruction that actually saves the word on the stack, reducing the required instruction overhead.

Making this all work requires that we introduce a few new instruction opcodes. As mentioned above we will need an instruction to save the SLAT bit-vector words. We call this instruction `sbv` for "save bit vector." It takes a single immediate operand specifying which bit-vector word to save. In order for callee-save store and load instructions to query the bit-vector, they need their own opcodes to distinguish them from ordinary memory operations. We add the `spill` and `reload` opcodes to do this. Other than the bit-vector and address redirection functions, they operate like normal load and store instructions, including lookup in the SLAT for their address operands[4].

In addition to callee-save spill and restore instructions, the compiler may spill registers in the course of ordinary register allocation when high register pressure is encountered. Practically speaking, this should never happen on a machine with 256 registers, but we have implemented a contingency for this in the MIRV compiler. If a register is spilled due to register pressure, it will be loaded from memory before each use and stored out to memory after each definition. If the register was speculatively allocated, the spill and restore operations will have to update the SLAT so that the machine knows whether the data currently lives in a register or in memory. This makes such spill operations very expensive. Performing two[5] loads before each use and two stores after each definition is

[4]Of course, being storage locations for callee-save registers, these addresses should never be mapped in the SLAT along correctly speculated program paths

[5]one for the mapped address and one for the actual data

Param	Value					
Issue	out-of-order					
Width	8					
Fetch Buffer	32 Instructions					
IQ	256 Entries					
LSQ	128 Entries					
Store Buffer	64 Entries					
ROB	512 Entries					
Branch Predictor	McFarlan Hybrid 2K 11-bit local history 13-bit global history 4-way 4K BTB 16 entry RAS 3 cycle mispredict penalty					
Function Units	**Integer**		**Floating Point**		**Memory**	
	ALU	4	ALU	4	DPorts	2
	Mult/Div	2	Mult/Div	1		
Cache	**L1 Instruction**		**L1 Data**		**L2 Unified**	
	Size	32K	Size	32K	Size	1M
	Assoc	2-way	Assoc	2-way	Assoc	4-way
	Line Size	32-byte	Line Size	32-byte	Line Size	32-byte
	MSHRs	32	MSHRs	32	MSHRs	32
	MSHR Tgts	16	MSHR Tgts	16	MSHR Tgts	16

Table 7.1: Simulation Parameters

unacceptable. Therefore, we have enhanced the MIRV compiler to undo speculative register promotion if such a register is spilled. Not only does this remove the SLAT overhead, it decreases register pressure without costing more memory operations than would have been necessary on a machine without support for speculative register allocation. As noted above, this contingency is necessary for correctness as well.

7.7 Methodology

In this section we describe our experiments to measure the effectiveness of speculative register promotion in the context of modern out-of-order microprocessors. As noted earlier these experiments use speculative register promotion only to address the separate compilation and side-effects issues. Overcoming the aliasing problem requires additional microarchitectural state and is covered in chapter 8.

The benchmarks used in this study are described in section 1.2. We run the SPEC95 and SPEC2000 integer and floating point benchmarks in our studies. All benchmarks were compiled by MIRV at optimization level -O2. This includes many classical transformations such as loop-invariant code motion, common subexpression elimination and copy propagation. It does not include transformations such as loop unrolling and function in-lining that tend to expand code size. The -O2 level includes non-speculative register promotion. A complete list of optimizations run at each level appears in appendix A. On top of everything implied by -O2 we add a speculative register promotion pass.

Machine parameters are provided in table 7.1. This is a fairy aggressive machine implementation. In particular we have provided many function units to remove structural hazard delays from the experiments.

We present overall performance and cache traffic data for several configurations. A baseline run is given 256 integer registers to utilize. The lower 32 registers operate according to the MIPS System V ABI while the remainder of the registers are evenly spilt between caller- and callee-save sets. Alongside the baseline run we perform three experiments with speculative register promotion. The first is a run with perfect prediction of SLAT conflicts and register mappings. Thus we eliminate the overhead caused by out-of-order rename.

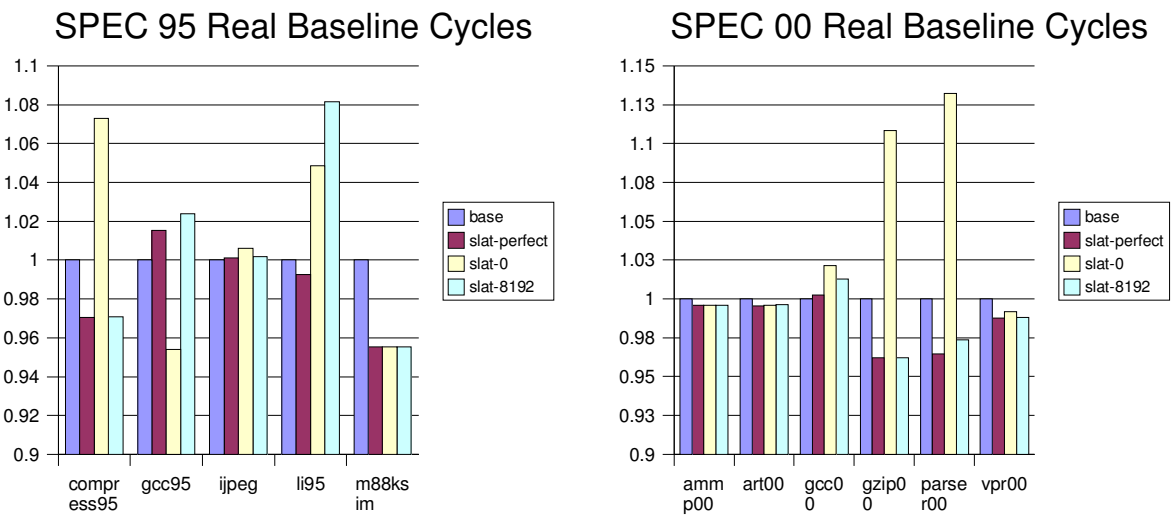
The next two experiments examine two prediction strategies. The first always predicts no conflict. The second predictor includes two tables: a table of two-bit saturating counters to predict whether there will be a conflict and a table of logical register tags to predict the mapped register if a conflict is predicted. Each table has 8K entries and is indexed by the address of the fetched instruction. For simplicity (and perhaps optimistically) we assume load and store instructions are predecoded in the cache so that we do not have to look up every instruction in the predictor.

```
void foo(void)
{
   spill $255
   li $255, 10
   for(...) {
      // ...
   }
   reload $255
}

int main(int argc)
{
   spill $255
   map $255, global
   for(...) {
      // Reference $255
      foo();
   }
   unmap $255
   reload $255
}
```

Figure 7.8: SLAT Aliasing Example

These three SLAT experiments are run under two machine models. The first model, which we call "Real," fully models the additional overhead instructions to spill and restore SLAT entries if a SLAT-mapped register is saved or restored by the compiler. The second model, which we call "Ideal," does not generate these extra instructions. Both models generate the extra operations to perform memory address checks for instructions that conflict or are predicted to conflict in the SLAT.

Our rationale is motivated by the example of figure 7.8. The register allocator in MIRV is heavily biased to always select registers from one end of the available pool when coloring the interference graph. It simply selects the first available register it finds for each local or promoted variable in the function. In the example, `global` has been promoted to register `$255` in `main`. Unfortunately, the loop counter in `foo` has also been allocated to register `$255`, forcing an expensive spill and reload of the SLAT entry. It is not even necessary that the data in `foo` be mapped in the SLAT.



(a) SPEC 95

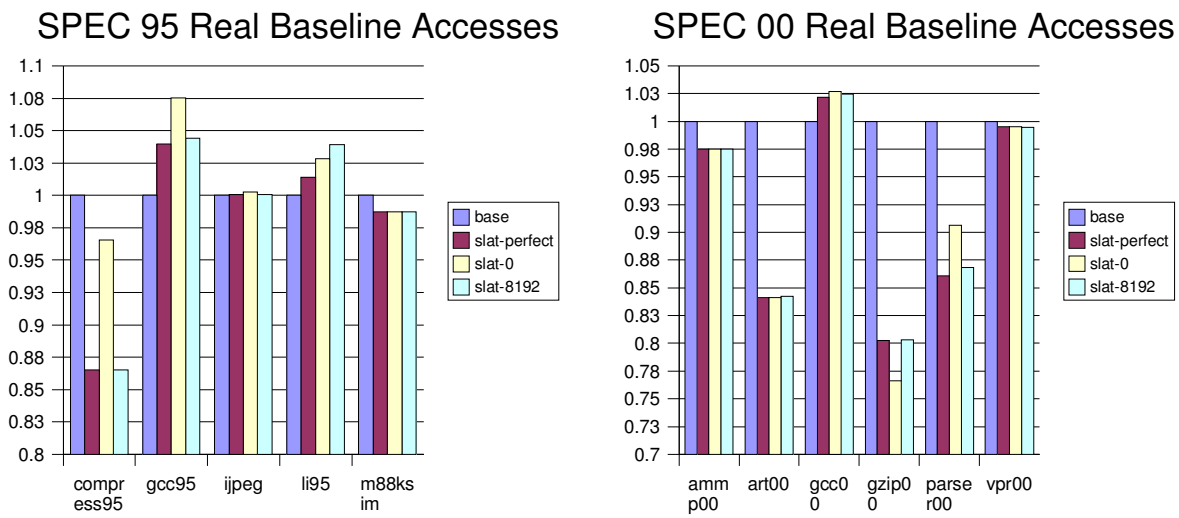
(b) SPEC 00

Figure 7.9: SLAT Performance

This overhead could be reduced with smarter register allocation policies. A simple strategy would be to randomly pick from the pool of available registers rather than always choosing the first available. This would tend to spread out register usage and reduce the number of such conflicts. The Ideal experiment models the best possible result of such manipulations.

7.8 Results

Figure 7.9 presents the performance of speculative register promotion including all overhead needed to spill SLAT entries and calculate memory addresses to verify SLAT predictions. Four bars are presented for each benchmark. The first is the relative performance of the benchmark without speculative register promotion. It is always one. The next three bars show the relative performance of speculative promotion to the baseline run. The second bar shows the performance with perfect prediction of SLAT conflicts. The third bar shows



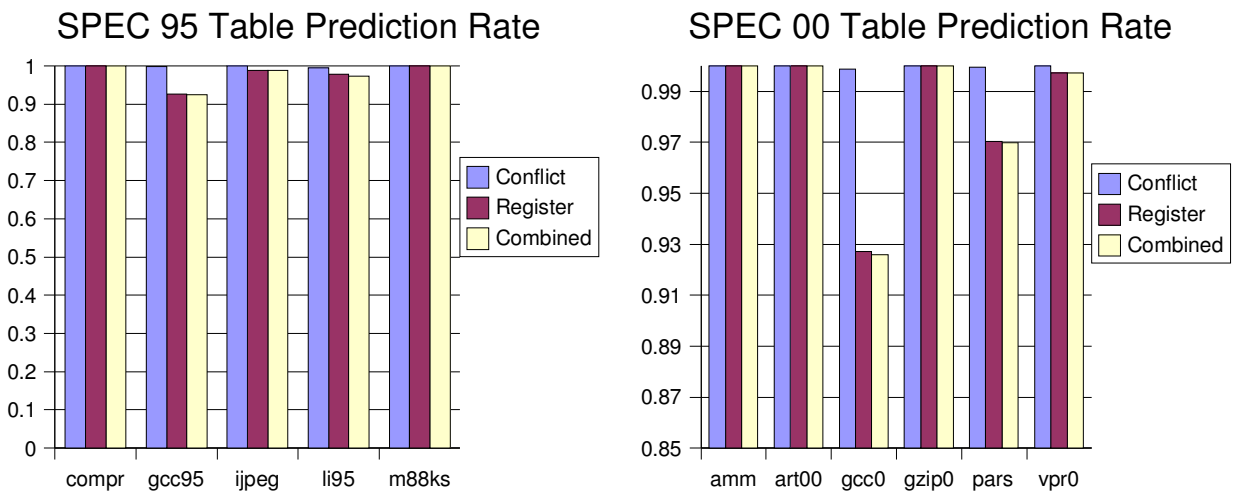
(a) SPEC 95

(b) SPEC 00

Figure 7.10: DL1 Accesses

the performance when always predicting no conflict and the last bar shows the performance with the simple table predictor of 8192 entries for the conflict counters and 8192 entries for target register prediction.

Performance on the benchmarks varies widely. Some, like compress95, m88ksim and gzip and parser, improve about 3%-4%. Other benchmaks, such as gcc95 and vortex00, show a performance degradation of 2%-3%. As noted by Postiff, speculative promotion tends to remove cache hits, leading to the conclusion that performance will only improve slightly due to the reduced dynamic instruction count [4]. Figure 7.10 shows the reduction in primary data cache accesses when speculative register promotion is used. The bars represent the same experiments as in figure 7.9. The number of DL1 accesses is reduced by up to 20% by speculative register promotion when using perfect prediction. Surprisingly, the predict-no-conflict scheme actually improves gzip even more, reducing the number of DL1 accesses by 23%. However, the 11% performance degradation shown in figure 7.9 indicates that this



(a) SPEC 95

(b) SPEC 00

Figure 7.11: SLAT Prediction Rates

is a poor tradeoff. Some benchmarks see up to a 4% increase in DL1 accesses.

Prediction rates for the table-based predictor appear in figure 7.11. Prediction rates range from 92% to nearly perfect. It is clear that SLAT conflicts and target registers are highly predictable. Our further experiments in this chapter and in chapter 8 will assume perfect prediction to gauge tradeoffs in SLAT design.

To help understand why performance and number of cache accesses can degrade when using speculative register promotion, we can calculate the expected overhead of speculative promotion. Each speculative promotion has an overhead cost to it. Because these promotions occur over function calls in loops, they must be placed into caller-save registers which require a register save at the top of the caller function and a register restore at the bottom. In addition, to perform the promotion the compiler must insert a `map` before the loop body and an `unmap` after it. The `unmap` is necessary even if the value is not written in the loop because the callee function may have written to it. As mentioned in section 7.6.1, when a SLAT-mapped register is saved in the function prologue and restored in the function

epilogue, the data must be written to or read from the home location and the SLAT entry must be saved to the stack location specified in the instruction.

Given all of the above overhead components and assuming probability ρ that a particular register save and restore will hit in the SLAT, we can express the expected overhead for each promotion p seen by the function as:

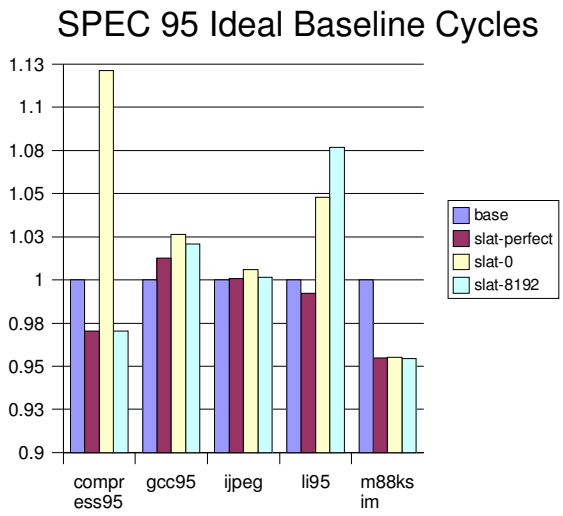
$$Overhead_p = 4 + 2\rho \tag{7.1}$$

Equation 7.1 calculates the overhead as the four extra memory instructions inserted by the compiler plus the two extra memory operations needed to save and restore the SLAT entry if the `spill` and `reload` instructions operate on a SLAT-mapped register minus the number of memory references converted to register references in the loop. Given this overhead and the number of promoted references in the loop N_p, the break-even number of loop iterations i_p required to overcome the overhead is:

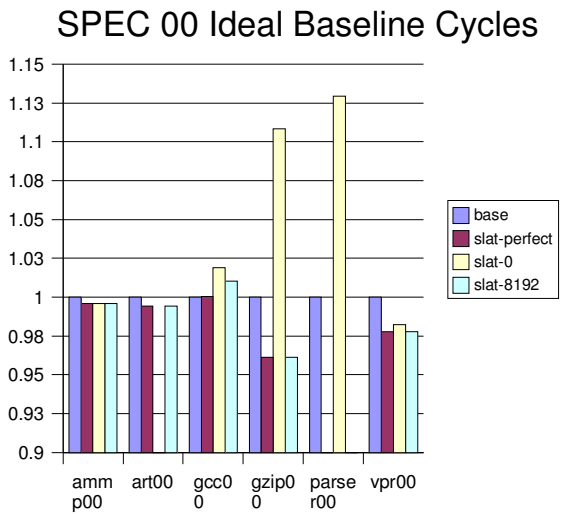
$$i_p = \frac{Overhead_p}{N_p} = \frac{4 + 2\rho}{N_p} \tag{7.2}$$

For example, if one memory reference is converted to a register reference in the loop and the save and restore do not affect any SLAT-mapped registers, it will require four loop iterations to recover the overhead of the promotion.

We ran experiments in which the simulator did not count the extra memory accesses needed when saving and restoring registers mapped in the SLAT. These results appear in figures 7.12 and 7.13. Both performance and number of cache accesses improves about 1%-2%, indicating that this overhead explains some of the performance losses but not all. Section 8.6 of chapter 8 further explores possible reasons for these losses.

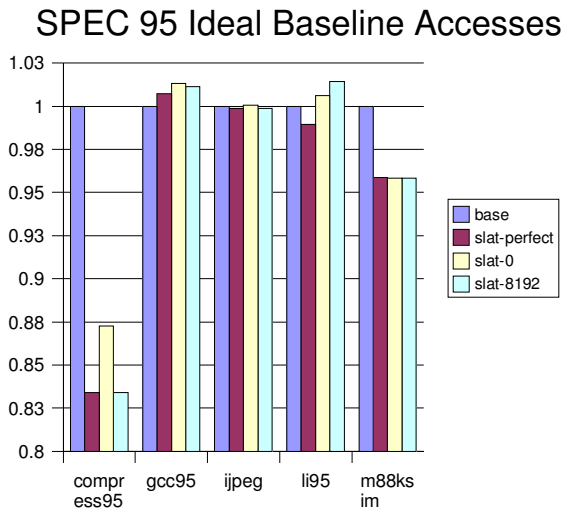


(a) SPEC 95

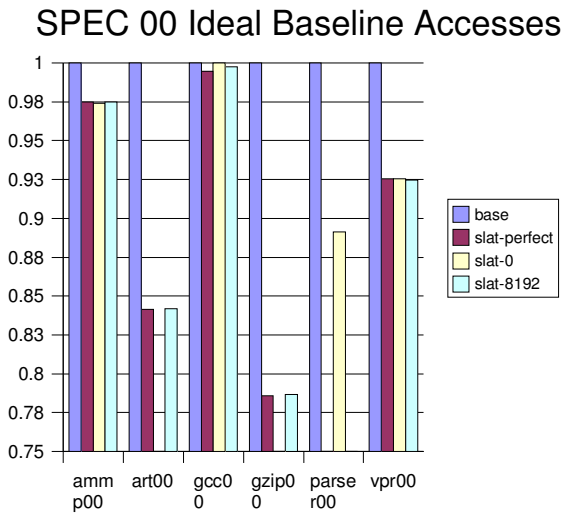


(b) SPEC 00

Figure 7.12: SLAT Performance



(a) SPEC 95



(b) SPEC 00

Figure 7.13: DL1 Accesses

7.9 Previous Work

There is a wealth of literature covering the areas of register allocation or use of other fast memory architectures to improve machine performance. We present this work in four major categories: compiler transformations that do not require architectural support, special architectural structures and cooperative techniques. Following this presentation we examine other work related to the SLAT implementation presented in this chapter.

Register allocation is important for a number of reasons. The processor-memory gap has continued to widen as improvements in processor architecture and fabrication processes continue to push the boundaries of clock frequency [86]. Though large caches have been made possible by high levels of chip integration, the much sought after wide issue rates have not been observed [59].

It is well-known that on average, more than a third of all instructions executed in a RISC-style processor perform memory operations [86]. Others have reported an even greater percentage for register-limited CISC architectures such as the Intel IA32 [87].

Many studies of optimal register set size have been performed. Mahlke, *et al.* explored how register file size affects the efficiency of multiple-issue processors [88]. This work argues for larger register sets to reduce the cost of spill code in the presence of aggressive compiler transformations, though at 24 compiler-allocatable registers was determined to be adequate for the majority of situations. A later study increased that number to between 70 and 128 [89]. Bradlee, *et al.* report that a register file of size 32 is sufficient provided the compiler generates code efficiently [90]. Sites argues that many more registers than are currently available today can be utilized effectively [85]. This array of contradictory results was noted by Postiff, *et al.* in a study of register utilization in integer codes [91].

7.9.1 Compiler Transformations

The most common register allocation technique in use today is the graph coloring method of Chaitin which was extended by Chow and Hennessy and later by Briggs in his dissertation [92, 93, 94, 95]. Traditionally the algorithm operates in an intra-procedural manner. Wall explored the possibility of allocating global variables to registers for their entire lifetime [6]. Chow presents another approach in which procedures are processed by the register allocator in a depth-first manner, allowing better register selection to reduce the overhead of callee- and caller-save register saves an restores [96]. In this work Chow also explores the shrink-wrapping transformation to places saves and restores only over regions where the registers are used, saving redundant operations on other program paths. Shrink-wrapping can help reduce the overhead component of speculative register promotion.

Cooper and Lu studied a register promotion algorithm similar to that used in MIRV [97]. Sastry and Ju propose an algorithm based on SSA form [98]. Neither of these studies considered speculatively allocating data to registers.

7.9.2 Architectural Structures

A number of novel architectural features have been proposed to combat the memory bottleneck. The Cray-I supercomputer used a multi-level register file to provide a large amount of fast storage. The primary register set is small and fast while the secondary set is large and slower. The compiler is responsible for managing the placement of data, effectively using the primary register set as a compiler-controlled cache [99]. The idea of providing an additional compiler-controlled memory space for long-lived values was extended by Cooper and Harvey [100]. Their compiler-controlled memory serves as a repository for registers spilled as a result of excessive register pressure.

A different approach to register allocation was taken by the Bell Labs C Machine [101]. This architecture provides no registers at all. Instead, special hardware exists to dynamically map stack references to a special stack cache.

Lee, *et al.* proposed a stack value file that operates in parallel with the primary data cache [102]. This value file is specially designed to use the special semantics of stack allocation to improve performance.

In order to support large numbers of registers, Postiff, *et al.* leveraged existing architectural support for out-of-order instruction issue to implement register caching for large logical register files [103]. We assume that a similar structure can be used to implement the logical register file of 256 registers used in this study.

7.9.3 Cooperative Techniques

The work in this chapter is an update to the original speculative register promotion study performed by Postiff, *et al.* [49]. This previous study primarily focused on instruction counts and did not consider the microarchitectural impacts of a SLAT implementation on a modern out-of-order microprocessor. In this chapter we have noted the out-of-order rename problem and proposed a speculation method to overcome it. In addition, the work of this chapter adds the spill and reload overhead required by registers mapped in the SLAT. Finally, we proposed an application binary interface for the SLAT.

The SLAT is closely related to the Advanced Load Address Table of the Intel IA64 architecture [38]. The ALAT is primarily used for code scheduling purposes to move loads above potentially conflicting stores. In addition, it can be used to speculatively promote loads out of loop constructs in the presence of potential store conflicts provided that the load is guaranteed to execute. Additional hardware exists that allows static control speculation to

move loads above branches. The ALAT watches all store operations and notes conflicts with promoted loads. Misspeculation recovery is performed in software by compiler-generated routines. Because the SLAT watches load operations in addition to stores, it allows full register promotion of loop-variant data. The drawback is that while the ALAT can lose information and trigger unnecessary fixups the SLAT must retain all mapping information in order to ensure program correctness. The ALAT is a direct descendent of the work on the Memory Conflict Buffer by Gallagher, *et al* [104].

To support efficient execution of the register-limited Intel IA32 programs on a RISC architecture, the Transmeta Crusoe processor includes special instructions to allow speculative register allocation of automatic variables [105, 106]. The `load-and-protect (ldp)` instruction marks a region as "protected" and the `store-under-alias-mask (stam)` instruction checks store operations for conflicts with e protected region. If a conflict is detected a software fixup routine is triggered. While this strategy requires that memory always remain synchronized to the register data, speculative register promotion with the SLAT allows the memory location to lag the value in the register until an `unmap` is performed. In addition, while the Crusoe code morph software executes at run-time, the SLAT allows static speculative register allocation. Finally, SLAT conflicts are entirely resolved within the hardware, making software traps unnecessary.

7.9.4 Other Related Work

Speculative register promotion can be seen as an alternative view of load speculation [107]. Specifically, the SLAT allows static prediction of load-store dependencies, analogous to the work on memory renaming and dynamic dependence prediction [83, 108, 109]. In the case of speculative register promotion, the communication conduit through which dependent

loads and stores is allocated ahead of time by the compiler: the register used as the target of a speculative promotion operation.

In our experiments a mispredicted SLAT conflict/register pair causes a flush of the entire pipeline behind the misspeculated operation, similar to the way branch mispredictions are handled. Selective recovery is a technique to reduce this penalty.

The fact that selective recovery is used on modern microprocessors affirms the viability of this technique to support speculative register promotion on an out-of-order machine [110]. The Pentium 4 architecture uses a form of data speculation to issue load-dependent instructions assuming that the load will hit in the cache. If the load misses, a replay mechanism re-executes only those instructions dependent on the load. Such mechanisms can be used to reduce the penalty of SLAT conflict mispredictions.

7.10 Conclusion

In this chapter we have demonstrated the viability of speculative register promotion using the SLAT on a modern, pipelined, out-of-order microarchitecture. Overall performance improvements are quite small, but the reduction in cache bandwidth can be significant. This is expected given that previous studies indicate speculative promotion primarily removes cache hits. Some benchmarks experience a loss in performance and an increase in cache bandwidth. This is due to the overhead of the speculative promotion. Clearly there is room for compiler heuristics to improve. Section 8.7 of chapter 8 presents additional avenues for future work.

CHAPTER 8

Expanded SLAT Architectures

8.1 Introduction

Chapter 7 identified several difficulties the SLAT presents for modern out-of-order microprocessors and described how the SLAT can be integrated into such designs. In addition, an ABI was prototyped to show one way the SLAT could be managed in a production environment.

This chapter describes several enhancements to the SLAT architecture to increase its utility and the applicability of the speculative register promotion transformation. One of our initial motivations for designing the SLAT was the static alias analysis problem. Our experience has taught us that alias analysis is an extremely difficult problem and obtaining good results requires very complex algorithms and time-consuming compiler debugging. Our previous work failed to make use of the SLAT in the case of aliasing. Our initial studies concentrated on overcoming the side-effect problem, particularly with respect to global variables. Indeed, Postiff's dissertation explains that the benefits of the SLAT are roughly equivalent to what is achievable with whole-problem global variable register allocation [4].

The SLAT enhancements presented in this chapter provide the mechanisms needed to

realize our initial vision of the SLAT as a tool to overcome the side-effect *and* aliasing problems. These enhancements are non-trivial, requiring some changes to the machine microarchitecture and the ABI described in section 7.6 of chapter 7. Therefore, we conduct several experiments to gauge the utility of the additional register promotions possible with the enhanced SLAT.

8.2 The Problem

In this section we explain why the SLAT architecture of chapter 7 is insufficient to handle the aliasing problem. In addition to describing the problem itself we examine the interplay between the compiler and the machine architecture to gain some insight about the form of a possible solution.

8.2.1 The Problem

Recall that the SLAT associates a register with each memory address entered into it. The compiler statically inserts instructions into the program to speculatively load data into a register and map its address in the SLAT. As long as the register is mapped to the address in the SLAT any conflicting memory references will be redirected to the register.

This works fine as long as there is a one-to-one mapping between memory addresses and registers. Consider the case of figure 8.1. The compiler cannot know statically whether `ip` points to `x` or `y`. Within the loop body it would like to promote `x` to a register because register allocate cannot consider it a candidate due to the fact that its address has been taken. It would also like to promote `*ip`.

Figure 8.2 shows the result of speculative register promotion on `x` and `*ip`. Notice that `x` has been promoted to register $4 and `*ip` to register $3. If we assume that `ip` points to

```
int main(int argc)
{
  int x, y;
  int *ip = &x;

  if (argc > 1) {
    ip = &y;
  }

  for(x = 0; x < 10; ++x) {
    *ip += x;
  }

  printf(``*ip = %d\n'', *ip);
  return(0);
}
```

Figure 8.1: SLAT Aliasing Example

```
main:
        sw $31,-4($sp)
        subu $sp,$sp,56
$L47:
        addu $5,$sp,44
        li $2,1
        slt $2,$2,$4
        beq $2,$0,$L50
$L49:
        addu $5,$sp,40
$L50:
        sw $0,44($sp)
        mapw $4,44($sp)
        mapw $3,($5)
$L51:
        addu $3,$3,$4
        addu $4,$4,1
        li $2,10
        slt $2,$4,$2
        bne $2,$0,$L51
$L52:
        unmapw $3,($5)
        unmapw $4,44($sp)
        la $4,$__mirv_pack.m1.206
        lw $5,($5)
        jal printf
        move $2,$0
$L48:
        addu $sp,$sp,56
        lw $31,-4($sp)
        j $31
```

Figure 8.2: Speculative Promotion of Potential Aliases

x during some run of the program, we will have violated the one-to-one mapping assumed above. The SLAT will contain two entries with the same memory address, each mapped to a different register. If a conflicting memory operation occurs, the machine will not know which register has the most up-to-date value. Stated another way, each update of registers $3 and $4 should communicate the value to the other register to maintain coherence. In essence we have transferred the aliasing problem from the core memory system to the register file.

This phenomenon has been described before, most thoroughly in the work on CRegs [111, 112, 113]. The CRegs implementation assumed a register file organized into clusters such that registers in a single cluster could be simultaneously updated. The compiler could then allocate aliased data into these registers as long as all references that potentially alias each other are placed into the same register set.

This multiple update requirement places an extra burden on a piece of hardware that is already taxed by current aggressive wide-issue out-of-order processor core designs [114, 115, 116]. Our goal with the SLAT is to eliminate this parallel update, reducing the complexity and cost of the register file. Our studies from chapter 7 simply disallowed promotion of any piece of data through more than one name. This guarantees that the same address is never mapped multiple times within the SLAT as long as registers are properly spilled and restored in the function prologue and epilogue.

8.2.2 A Bird's-Eye View

At this point we seem to be stuck. It is not possible for the compiler to know whether two names actually alias each other at run-time. Therefore it cannot guarantee that promoting those names to different registers will result in different address mappings at run-time. On the other hand, if the compiler assumes the names are aliases and promotes them to the

same register, incorrect results will be produced if the names are in fact not aliases at runtime. Worse yet, because the register names are hard-coded into the instruction words themselves, there is little hope of hardware help without complex register-to-address reverse mappings in addition to the SLAT itself.

If we step back a moment to examine what is going on in the compiler we can move toward a possible solution. Recall the register allocation process finds a mapping from symbolic program name to a particular machine-architected register name. Different symbolic names that are simultaneously live must be given different architectural register names to prevent data corruption. Symbolic names that are equivalent must be given the same architected register name. In particular, if a pointer is known to only point to one piece of data the result of dereferencing it may be placed in the same register as the "canonical" name. This is usually accomplished indirectly by replacing the dereferenced pointer with the canonical name.

In our example above, if the compiler statically knew that `ip` points to `x` at all times, it would simply allocate `*ip` to register $4. On the other hand, if it knew that `ip` never pointed to `x` it would maintain the mapping of `*ip` to register $3. Although the compiler cannot have this knowledge statically, the processor *does* have it dynamically. It available right at the point of the second `map` instruction in figure 8.2. Given this availability of dynamic aliasing information, we can construct a solution to the aliased promotion problem.

8.3 Design

In an ideal, oracle environment, the compiler would have made the correct decision to assign `*ip` to register `$3` or `$4` depending on which way it "knew" the pointer would point. A dynamic compiler could make this decision each time through the code at the additional

A	$4
B	$13
B	$14

(a) SLAT Contents

$13	p43
$14	p43
$15	p128

(b) RAT Contents

Figure 8.3: Physical Register Sharing

cost of recompilation or code generation. In this section we propose a pipeline enhancement to perform the proper naming dynamically.

Current out-of-order microprocessors already contain a component to map one set of names to another: the register renaming engine near the front of the pipeline. We can use this to our advantage to solve the name aliasing problem. If we view the register renamer as a preprocessor on the program text, we can seize the opportunity to "re-write" parts of the program that are found to be incorrect. In essence, we use the register renaming hardware as a simple dynamic translator or compiler.

One obvious use of the register renamer to solve the speculative promotion aliasing problem is to simply force the renaming hardware to assign the target register of a `map` instruction the current physical register used by the already-mapped alias. This is illustrated in figure 8.3. Unfortunately, this complicates the commit state of the processor pipeline. A common scheme for freeing physical register tags after commit requires that a physical tag can only be released after the `next` write to the same logical register commits [117, 118]. This guarantees that all users of the data referenced by that physical tag have been committed and therefore that tag is no longer needed. With the scheme of figure 8.3, we may have two or more logical registers mapped to the same physical register. Determining when all uses of those logical registers have been committed is much more complex.

```
int main(void)
{
  map r5, global
  map r6, *p
  for(...) {
    r5 = r5 + 1
    r6 = r6 + 1
    foo()
  }
  unmap global, r5
  unmap *p, r6
  return(0);
}
```

Figure 8.4: Speculative Promotion Alias

global	r5
global	r6
A	r12

(a) SLAT Contents

r5	p61
r6	p61
r13	p128

(b) RAT Contents

Figure 8.5: Physical Register Sharing Difficulty

There is another, more subtle and more serious problem with sharing physical registers in this way. Figure 8.4 shows a small piece of code with an (assumed) aliasing situation. Figure 8.5 shows the state of the slat and the RAT after the first loop iteration. Imagine that program control iterates around the loop. Upon reaching the redefinition of `r5`, a new physical register will be assigned to `r5`. When we enter the next statement and read `r6` we need to see the new value assigned to `r5` because `r6` aliases it. However, `r6` is still mapped to `p61`. When `r6` is updated, `r5` will also need to see the new value.

This situation sheds some light on how to determine when all uses of a physical register have been committed. It seems from this example that `any` write to a member of the "register alias set" that commits will free the physical tag. Unfortunately, the situation of figure 8.4 has pushed the problem to the front of the machine. Whenever any register in the alias set has been renamed, all registers in the set must be renamed at the same time so that later instructions that reference those registers will receive the correct value. Determining the members of such sets and update all of the RAT entries for them is equivalent to updating register sets in CRegs, which we are trying to avoid [111]. We have moved the aliasing problem into the RAT update hardware.

8.3.1 The Logical Rename Table

Recall that in section 8.2.2 we viewed the alias problem from the perspective of the compiler and noted that if it had perfect knowledge it would simply assign the correct (logical) register names appropriately. This knowledge is readily available at runtime. To make use of it and solve our register aliasing dilemma, we propose a new piece of hardware called the *Logical Rename Table* (LRT).

The idea of the LRT is simple. Rather than complicate the RAT logical-to-physical

r4	r4
r5	r5
r6	r6

LRT Contents

r4	p43
r5	p12
r6	p128

RAT Contents

(a) Map `r5`

r4	r4
r5	r5
r6	r5

LRT Contents

r4	p43
r5	p12
r6	p128

RAT Contents

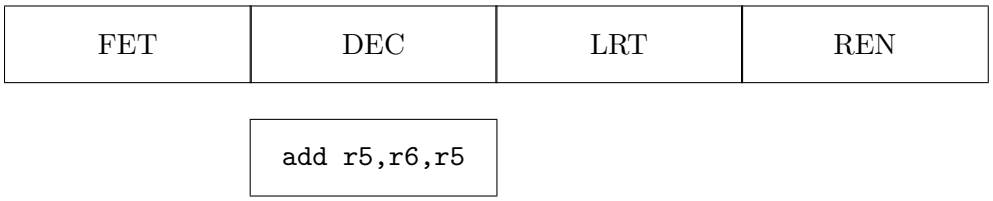
(b) Map `r6`

Figure 8.6: LRT Operation

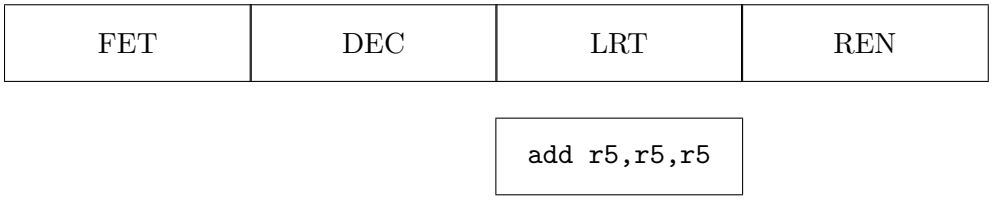
register mapping hardware, control the aliasing within the logical names themselves. In other words, the LRT holds a state separate from the rename hardware proper to resolve aliasing issues. The machine can detect an aliasing situation upon execution of a `map` instruction. If the source address already exists in the SLAT[1] then we have encountered an aliasing situation. At this point, the LRT sets up a mapping from the logical register targeted by the `map` operation to the logical register currently mapped in the SLAT. All further instructions that reference the most recently mapped register will be rewritten to reference the register already in the SLAT.

Operation of the LRT on the code of figure 8.4 is illustrated in figure 8.6. Figure 8.6(a) shows the state of the RAT and LRT after `r5` is mapped. Figure 8.6(b) updates the LRT with the appropriate information when `r6` is mapped. Finally, figure 8.7 shows how

[1]In the same call frame as explained in section 7.4.



(a) End of Decode



(b) End of Logical Rename

(c) End of Physical Rename

Figure 8.7: LRT Pipeline Flow

instructions that use and define registers `r5` and `r6` flow through the renaming stages of the pipeline.

The LRT separates the complication of register aliasing from the RAT. The RAT and the rename hardware associated with it remain unchanged. In effect, the LRT rewrites the program binary before the out-of-order mechanism ever sees it. To the point of view of the rest of the processor, the LRT simply provides an ordinary static program image.

This is not without its complications, however. Just as SLAT conflicts require a pipeline flush and restart, so too do aliasing situations. Fortunately, all of the required hardware is already available. Because `map` and `unmap` instructions are simply special load and store instructions, they must query the SLAT to resolve any conflicts. For the design of chapter 7, these can only conflict with addresses mapped in previous call stack frames. In those cases the machine will have to flush and restart the pipeline due the the out-of-order rename problem described in section 7.5. The LRT adds the possibility that map and unmap operations will conflict with addresses mapped in the same call frame. The corrective action is the same: flush and restart the pipeline. However, in addition the `map` operation will update the LRT to point the target register to the aliased logical name. The `unmap` instruction will remove the LRT entry.

8.3.2 ABI Impacts

Section 7.6 of chapter 7 describes how register spill and reload operations in the function prologue and epilogue affect the SLAT. The LRT adds a slight twist to the `spill` and `reload` operations. Assume for a moment that no such instructions are present in the program. In this case, the first map of `global` to `r5` in figure 8.6 will assign physical register `p61`. When `*p` is mapped, the LRT will add an entry mapping `r6` to `r5`. Because the compiler

can construct speculative promotions and demotions in a nested fashion, we can guarantee that any explicit unmap of `r6` will occur before the unmap of `r5`. Therefore the LRT can always hold the necessary information to maintain the correct mappings.

Unfortunately, function prologue and epilogue code disturbs our Garden of Eden. At any point (perhaps in the execution of `foo`) register `r5` may be spilled to the stack. Separate compilation prevents the compiler from managing such spills and reloads in a nested fashion. If `r5` is spilled a decision must be made about what impact that will have on `r6`. The data for both registers lives in physical register `p61`. Upon the spill, that data is written out to the home location `global` and `r5` is later given a new, unrelated value. If the LRT were to maintain the `r6` to `r5` mapping any read of `r6` would obtain an incorrect value.

One option is to remove the mapping from the LRT. This effectively removes `global` from the SLAT completely, meaning no memory references will be redirected to registers. Because such spill code can only occur in code other than the function where speculative promotion was performed[2], no memory operations will have been statically rewritten to reference registers directly. Thus correctness is maintained with this approach. With this policy we refer to `r5` as the *master register*. Such a *master register spill* removes the data from the SLAT, effectively reversing the speculative promotion temporarily.

Another option is to change the SLAT entry to reference `r6` instead of `r5` and updating the LRT appropriately. This requires assigning a new physical register to `r6`. Fortunately, we have such a register in `p61`. Presumably the compiler only spilled `r5` because it was about to redefine it in the body of `foo`. Thus any such spill can be considered a last use of `p61`, freeing it for reassignment to `r6`. In fact `r6` still contains the value of `global` so we really have not changed the data state of the machine at all. We have simply rewired the

[2]The compiler can guarantee that neither `r5` nor `r6` are spilled within the static "window of speculation."

alias mapping. When `r5` is reloaded, it will find `r6` mapped in the SLAT and an LRT entry mapping from `r5` to `r6` can be created. Of course this will upset the unmap balancing in `main`. However, when `r6` is unmapped we simply reverse the process above and point `r5` to `p61`.

8.4 Further Enhancements

In addition to the enhancements of section 8.3 to tackle the aliasing problem we wish to explore enhancements not studied in our original work. The original SLAT study restricted register makes to 32-bit data items exclusively. This was done primarily to simplify the compiler and simulator but it also makes the hardware designer's job easier. We wish to quantify the performance gains possible if a variety of data sizes can be mapped in the SLAT.

8.4.1 Architectural Impact

Mapping multiple-sized data in the SLAT presents a number of problems for the architect. The first is how to detect conflicts. Because non-word-sized data may be aligned at non-word addresses, additional size information must be stored in the SLAT and a more complex comparator must be employed. We assume (perhaps optimistically) that such a comparator can operate within the processor cycle time.

If a conflict is detected, appropriate measures must be taken to load or store the correct portion of the register data corresponding to the memory access. If the memory access is smaller than the size of the data item mapped, shifting and masking must be used to load or store the portion of the register data affected. The baseline SLAT also has this requirement. This problem becomes more frequent if non-word items are mapped in the

SLAT because routines such as `memcpy` tend to operate in word-size chunks. Any non-word data items mapped in the SLAT will require shifting and masking to maintain the correct data in the registers. This is potentially quite frequently needed for C++ code because default constructors are typically implemented with routines similar to `memcpy`.

8.4.2 Compiler Impact

Such mappings may also complicate the compiler. In the non-aliasing case, it will not be possible for multiple data sizes within the same word to be mapped simultaneously. However, if the LRT is used to providing alias mapping capability, the compiler's job is complicated. The primary problem is maintaining the correct data in the registers for non-word items. Arithmetic computation on such data require that the compiler perform shifts or bitmasks to maintain proper sign- and zero-extension. Because the LRT maps the data to a single physical register, such manipulations can destroy information needed by an larger-sized data items mapped to the same register. Therefore, we do not want to allow simultaneous mappings of multiple data sizes within a single word.

To implement this policy, we enforce the following restrictions in the compiler:

- No non-word item may be mapped through a union reference
- No item not of a pointer's "native" size may be mapped
- No non-word item may be mapped through an "abstract" pointer

The "native" pointer size mentioned in rule two refers to the size of the pointed-to object typed by the type of the pointer. For example, the "native" size of a `char *` is one byte and the "native" size of a `short *` is two bytes on the PISA architecture. The "abstract" pointer of rule three refers to a pointer that is determined to possibly point to anonymous,

`all`, `allAliased`, `allAliasedGlobal` and `ref` data[3].

The first rule ensures that we do not accidentally map aliased data of multiple sizes through the overlap provided by a union type. This is illustrated by the example of figure 8.8. If `halfword` is promoted in `main` and the byte references are promoted in `foo` we will map a byte and a halfword from the same address to different logical registers[4]. Restricting promotion of union data members to word-size items is a conservative but correct solution. We could promote `halfword` and not promote the byte union members but is impossible for the compiler to make such decisions in the general case.

An example of the restrictions imposed by the second rule appears in figure 8.9. In this example casting has been used to make a `char *` pointer point to a `short` data item. The references to the bytes in `foo` should not be promoted because `lower_short` and `upper_short` may have been mapped in `main`. In this case the byte pointers point to non-byte data items and so these dereferences fall under rule two, meaning they cannot be promoted.

Unfortunately, due to separate compilation, rules one and two are not sufficient to cover all cases of multiple-size aliasing. An example of the gap appears in figure 8.10. Assume that `main` and `foo` are in different compilation units. The basic problem is that in `foo` we do not have any idea of what `p` points to. In fact, in this example `p` points to a union member and thus the bytes accessed overlap the `halfword` accessed in main. If all references were promoted we would again see aliased data of data sizes places in several logical registers. Rule three is a blanket measure to catch this case. Pointer `p` will be determined to point to any global data and thus rule three kicks in for the byte accesses through the pointer.

[3]See section 3.2.5 of chapter 3.

[4]This example assume we use the LRT to handle multiple mappings of the same address across procedure calls.

```
union s {
  short halfword;
  struct c {
    signed char low_byte;
    signed char high_byte;
  } bytes;
} global = { 0 };

void bar(void)
{
}

void foo(void)
{
  int i;

  for(i = 0; i < 2; ++i) {
    global.bytes.low_byte ^= global.bytes.high_byte;
    global.bytes.high_byte ^= global.bytes.low_byte;
    bar();
  }
}

int main(void)
{
  int i;

  for(i = 0; i < 10; ++i) {
    global.halfword += i;
    foo();

    printf("halfword = %d\n", global.halfword);
    printf("low_byte = %d\n", global.bytes.low_byte);
    printf("high_byte = %d\n", global.bytes.high_byte);
  }

  return(0);
}
```

Figure 8.8: Multiple-size Aliases Through a Union

```
struct s {
  short lower_short;
  short upper_short;
} global = {0, 0};

void bar(void)
{
}

void foo(void)
{
  int i;

  char *byte_3 = (char *)(&global.upper_short) + 1;
  char *byte_2 = (char *)(&global.upper_short);
  char *byte_1 = (char *)(&global.lower_short) + 1;
  char *byte_0 = (char *)(&global.lower_short);

  for(i = 0; i < 2; ++i) {
    *byte_3 ^= *byte_1;
    *byte_2 ^= *byte_0;
    *byte_1 ^= *byte_3;
    *byte_0 ^= *byte_2;
    bar();
  }
}

int main(void)
{
  int i;

  for(i = 0; i < 10; ++i) {
    global.lower_short += i;
    global.upper_short -= i;

    foo();

    printf("upper_short = %d\n", global.upper_short);
    printf("lower_short = %d\n", global.lower_short);
  }

  return(0);
}
```

Figure 8.9: Multiple-size Aliases Through a Pointer

```
struct c {
  signed char low_byte;
  signed char high_byte;
};

union s {
  short halfword;
  struct c bytes;
} global = { 0 };

void bar(void)
{
}

/* We don't know p points to a union member. */
void foo(struct c *p)
{
  int i;

  for(i = 0; i < 2; ++i) {
    p->low_byte ^= p->high_byte;
    p->high_byte ^= p->low_byte;
    bar();
  }
}

int main(void)
{
  int i;

  for(i = 0; i < 10; ++i) {
    global.halfword += i;
    foo(&global.bytes);

    printf("halfword = %d\n", global.halfword);
    printf("low_byte = %d\n", global.bytes.low_byte);
    printf("high_byte = %d\n", global.bytes.high_byte);
  }

  return(0);
}
```

Figure 8.10: Multiple-size Aliases Through an Abstract Pointer

8.5 Methodology

For this study, we run four sets of experiments. The first is a cycle-accurate study of SLAT performance on a modern out-of-order processor. We maintain the size and aliasing restrictions of previous work. The goal is to measure the effect of the additional memory operations required to handling spills and reloads of speculatively promoted data and the additional address computation instructions needed to verify SLAT predictions for conflicting load and store instructions. We run a set of these baseline experiments with a perfect SLAT predictor, no SLAT predictor (i.e. always predict no-conflict) and an 8192-entry predictor to measure the impact of conflict mispredictions.

Our second set of experiments quantifies the advantage of allowing multiple-sized data in the SLAT. We do not consider promoting aliased data at this time. We simply wish to see what additional gains could be had from the "traditional" SLAT and renaming architecture.

We study the aliasing case in our third set of experiments. The hardware includes the LRT for alias renaming and the compiler is free to place potentially aliased memory references into the SLAT. We do not consider multiple-size data in this experience as we wish to isolate the benefit of the alias-capable SLAT architecture.

Our final set of experiments combines the new techniques. We allow aliased and multiple-sized maps into the SLAT with the set of restrictions presented in section 8.4.

8.5.1 Simulation Environment

We used the M5 simulator in all our experiments to model the SLAT and additional renaming hardware. All of the extra hardware-generated memory operations for spills and reloads of SLAT mapped-registers are simulated. The instructions to spill to- and from- the home memory location are not in any way dependent on the instructions to spill the SLAT

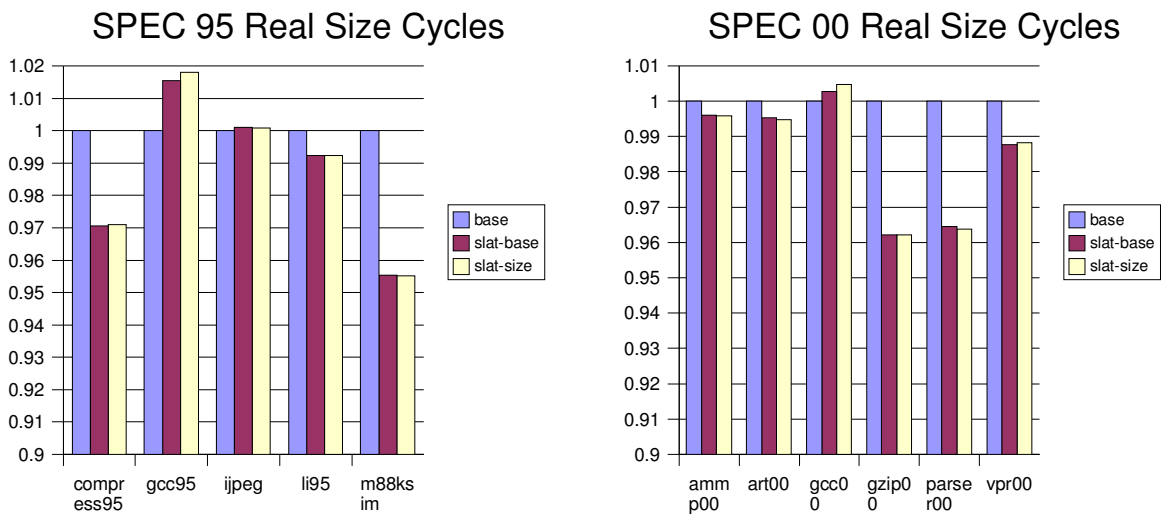
Param	Value					
Issue	out-of-order					
Width	8					
Fetch Buffer	32 Instructions					
IQ	256 Entries					
LSQ	128 Entries					
Store Buffer	64 Entries					
ROB	512 Entries					
Branch Predictor	McFarlan Hybrid 11-bit local history 2K local history table 13-bit global history 4-way 4K BTB 16 entry RAS 3 cycle mispredict penalty					
Function Units	**Integer**		**Floating Point**		**Memory**	
	ALU	4	ALU	4	DPorts	2
	Mult/Div	2	Mult/Div	1		
Cache	**L1 Instruction**		**L1 Data**		**L2 Unified**	
	Size	32K	Size	32K	Size	1M
	Assoc	2-way	Assoc	2-way	Assoc	4-way
	Line Size	32-byte	Line Size	32-byte	Line Size	32-byte
	MSHRs	32	MSHRs	32	MSHRs	32
	MSHR Tgts	16	MSHR Tgts	16	MSHR Tgts	16

Table 8.1: Simulation Parameters

data to the stack location.

Memory operations that are predicted to conflict in the SLAT are converted into register copies during the decode stage and an additional instruction representing the memory address calculation and lookup in the SLAT is generated. The register copy is not dependent on this additional instruction and thus the primary penalty of the check is the use of the adder to perform the address computation.

We modified M5 to recognize the new instructions required by the SLAT and implemented a functional model of the SLAT hardware. The machine parameters we used in our experiments are listed in table 8.1.



(a) SPEC 95

(b) SPEC 00

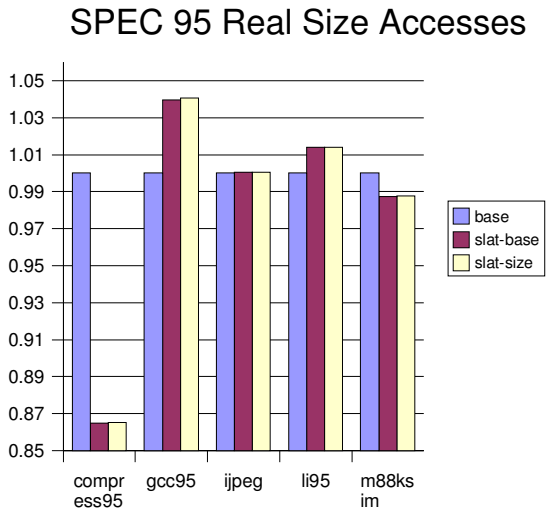
Figure 8.11: SLAT Size Performance

8.6 Experimental Results

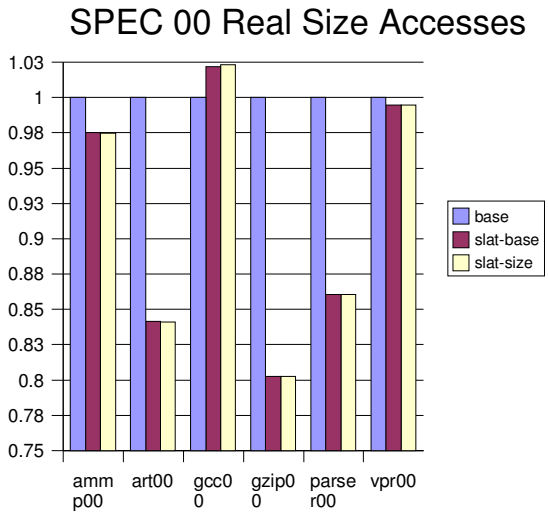
In this section we examine how the various enhancements to the SLAT presented in the chapter compare to the baseline SLAT of chapter 7. We individually examine the effects of allowing multiple sizes and aliases in the SLAT and then explore them in combination. All experiments are run assuming a perfect SLAT predictor.

8.6.1 Size

Figures 8.11 and 8.12 show the results for a SLAT that can handle multiple sizes. Figures 8.13 and 8.14 show the corresponding results if the register save/restore overhead is eliminated. Again we see that this overhead is negligible. Mapping multiple-sized data in the SLAT does not affect performance very much. For some benchmarks a slight improvement is observed while a degradation is observed for others. Most are not affected at all. This is not terribly surprising given the restrictions of section 8.4.2. Memory bandwidth is likewise

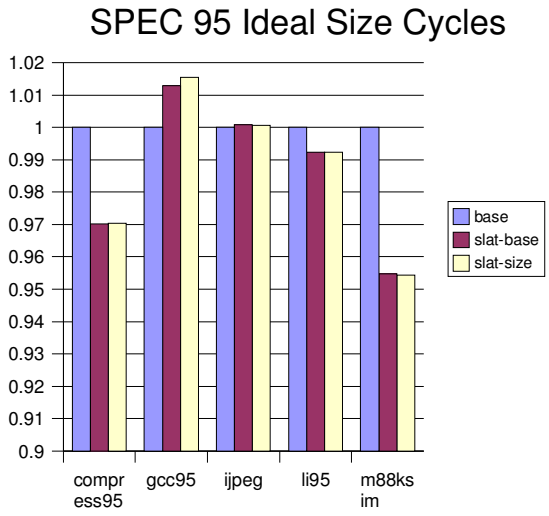


(a) SPEC 95

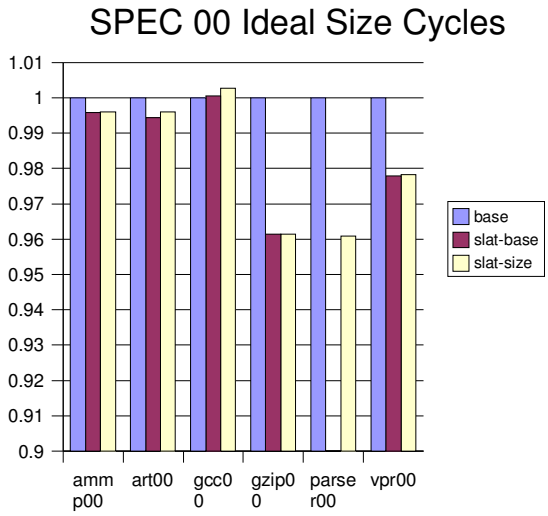


(b) SPEC 00

Figure 8.12: DL1 Size Accesses

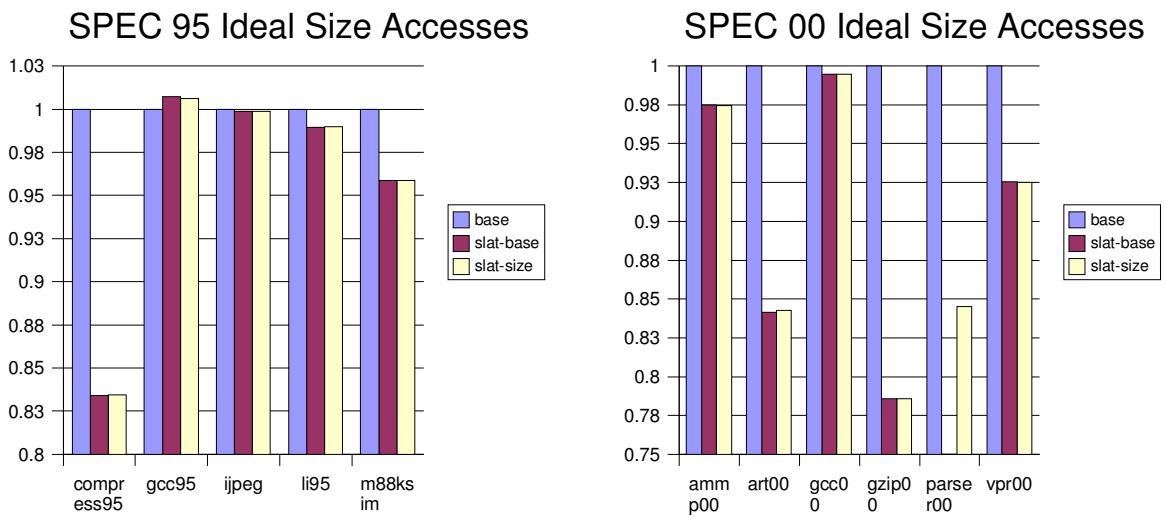


(a) SPEC 95



(b) SPEC 00

Figure 8.13: SLAT Ideal Size Performance



(a) SPEC 95

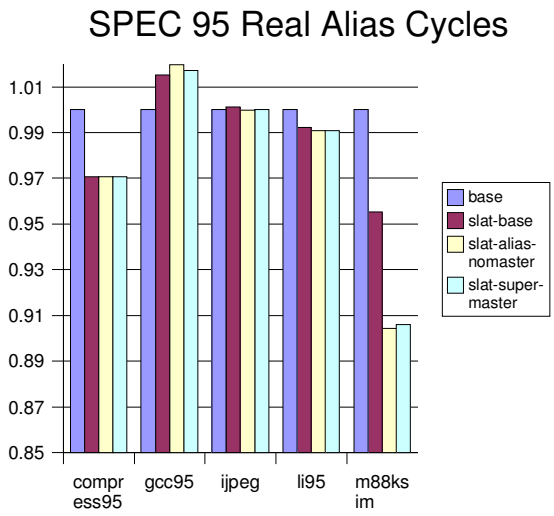
(b) SPEC 00

Figure 8.14: DL1 Ideal Size Accesses

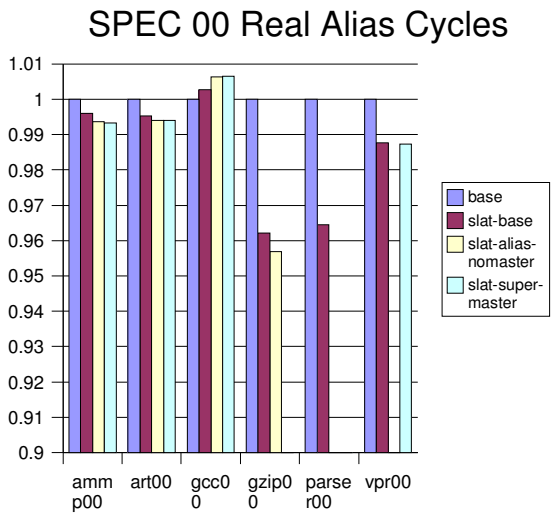
unaffected and thus we conclude that the SLAT size enhancements alone are not justified for these benchmarks.

8.6.2 Aliases

Figures 8.15, 8.16, 8.17 and 8.18 reflect performance and memory bandwidth when the LRT is added to the SLAT. The figures shows four bars for each benchmark. The first is baseline performance without the SLAT while the second repeats the baseline SLAT performance numbers from chapter 7. The third and fourth bars indicate the impact of master register tracking. The third bar shows performance of an architecture that can update the LRT when the master register is unmapped from the SLAT. One of the remaining registers in the alias set is chosen to become the target mapping of the relevant LRT entries. The last bar indicates the performance when the LRT is cleared when the master register is unmapped from the SLAT. This effectively removes the mapped address from

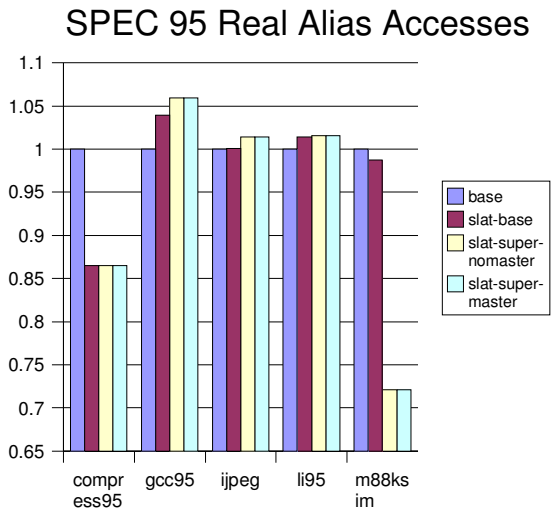


(a) SPEC 95

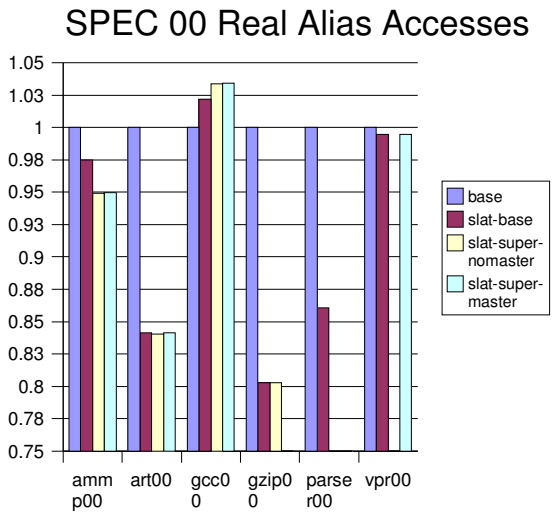


(b) SPEC 00

Figure 8.15: SLAT Alias Performance

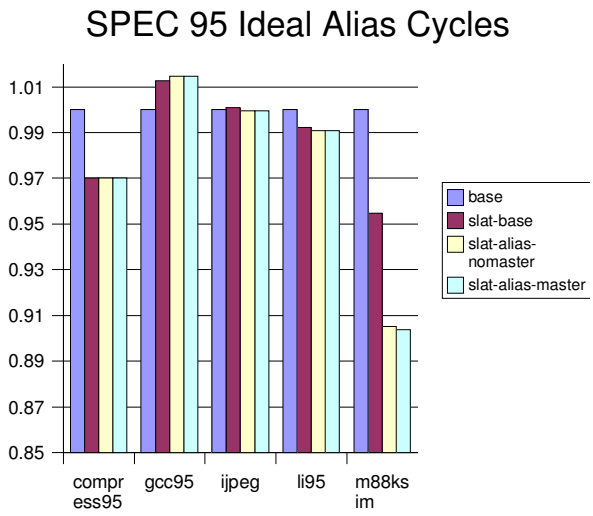


(a) SPEC 95

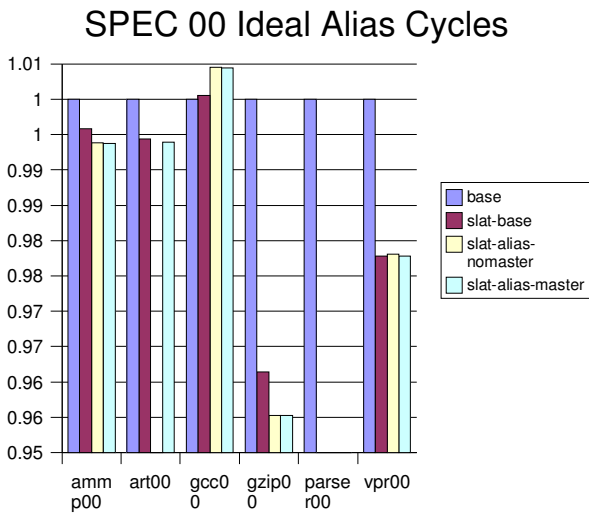


(b) SPEC 00

Figure 8.16: DL1 Alias Accesses

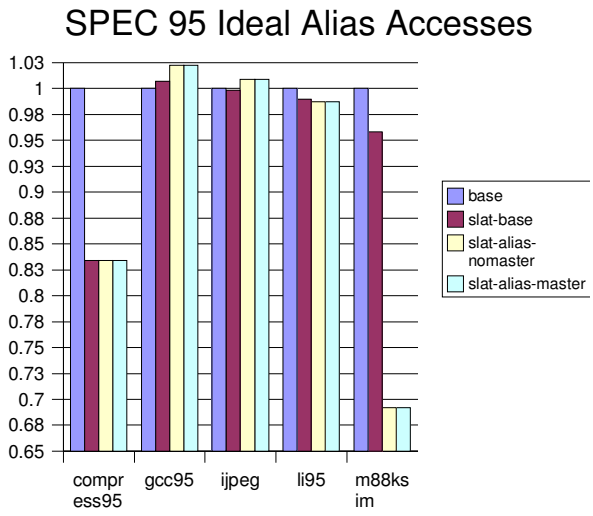


(a) SPEC 95

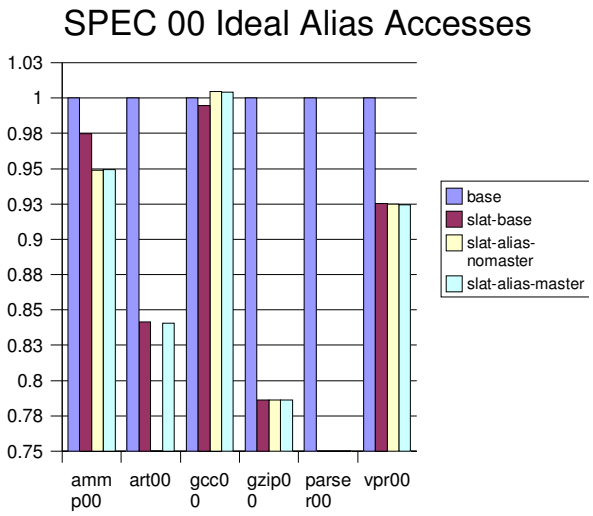


(b) SPEC 00

Figure 8.17: SLAT Ideal Alias Performance



(a) SPEC 95



(b) SPEC 00

Figure 8.18: DL1 Ideal Alias Accesses

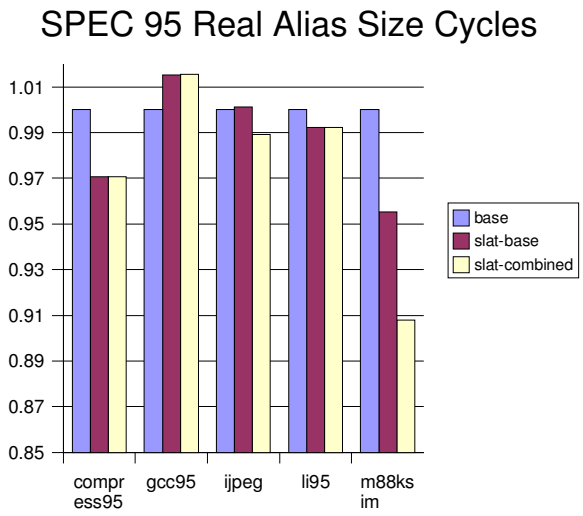
the SLAT. The policy appears to have very little impact but as would be expected the LRT update policy gives a slight benefit to most benchmarks. The "master spill" policy in fact *improves* performance slightly on m88ksim and gcc00. This is possible for two reasons: the register save/restore overhead component changes based on what is mapped in the SLAT and removing data from the SLAT and placing it in memory changes the cache access pattern, which may gain additional performance through prefetching due to the cache line size[5]. Overall performance improves about 4% for the m88ksim benchmark but overall the aliasing affect seems to be very benchmark dependent. None of the other benchmarks see much of a change.

When primary data cache bandwidth is considered, we see a whopping 27% reduction in data cache access for m88ksim. Again, the other benchmarks seem hardly affected. The `alignd` routine of m88ksim is one contributor to the improvement. This routine consists of a series of loops that manipulate word-sized data through pointer parameters. These parameters must be assumed by the compiler to alias each other and thus the data they point to is not eligible for register allocation. This is precisely the type of situation for which the LRT is designed and it performs well in this case. In fact the non-loop portions of the routine also contain code that manipulates the data pointed to by these parameters and thus the benchmark may benefit from speculative promotion over the entire function body. This would also reduce the `map`/`unmap` overhead.

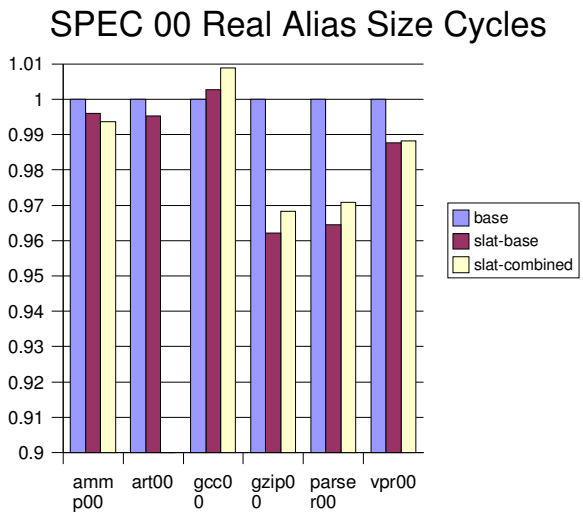
8.6.3 Alias and Size

Figures 8.19, 8.20, 8.21 and 8.22 indicate the effects of combined mapping of multiple-sized and aliased data in the SLAT. Again, we see that the size enhancements do not justify

[5]Recall that the SLAT usually maps accesses that would be cache hits.

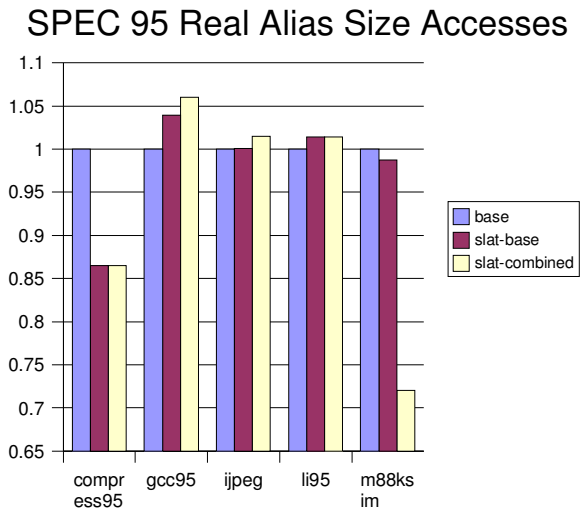


(a) SPEC 95

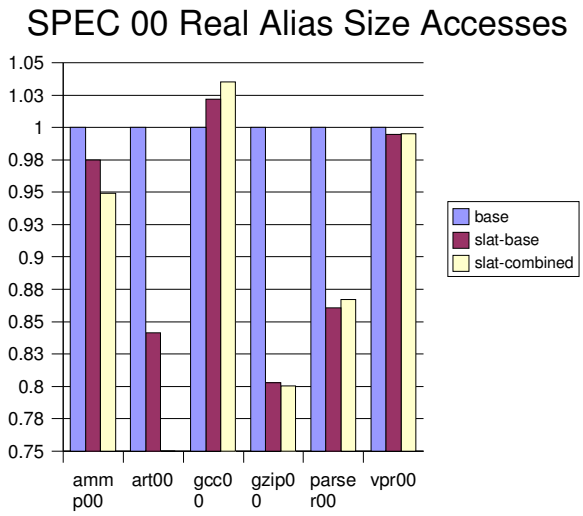


(b) SPEC 00

Figure 8.19: SLAT Combined Performance

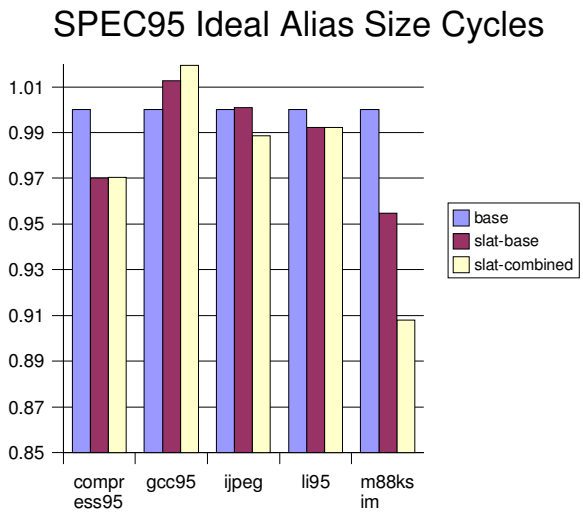


(a) SPEC 95

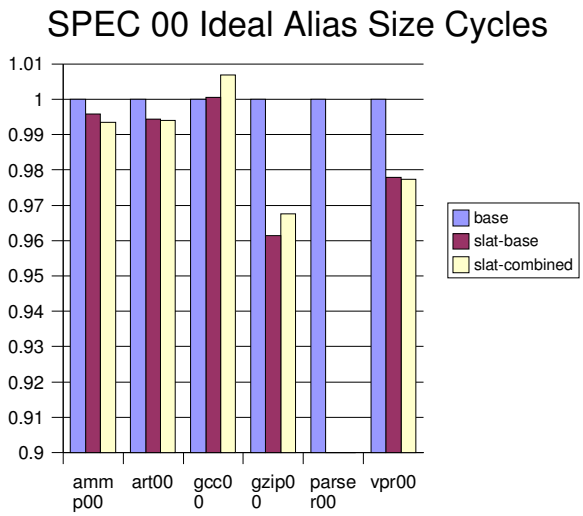


(b) SPEC 00

Figure 8.20: DL1 Combined Accesses

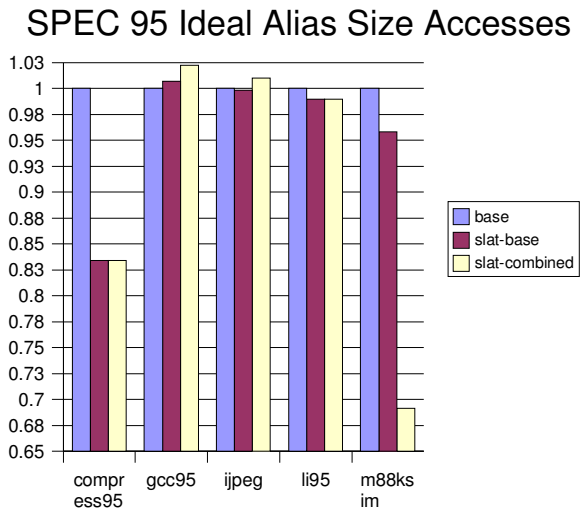


(a) SPEC 95

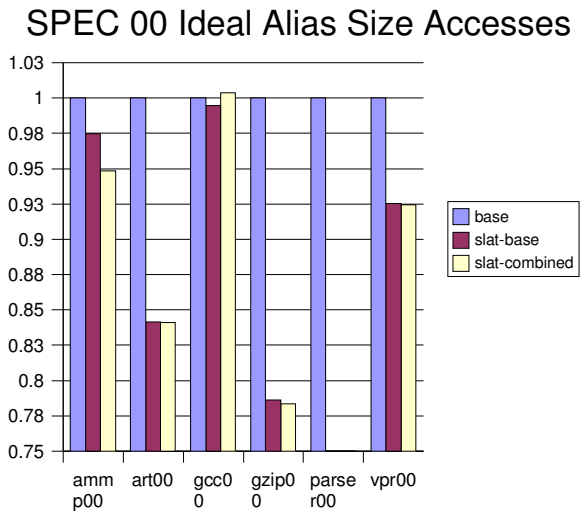


(b) SPEC 00

Figure 8.21: SLAT Ideal Combined Performance

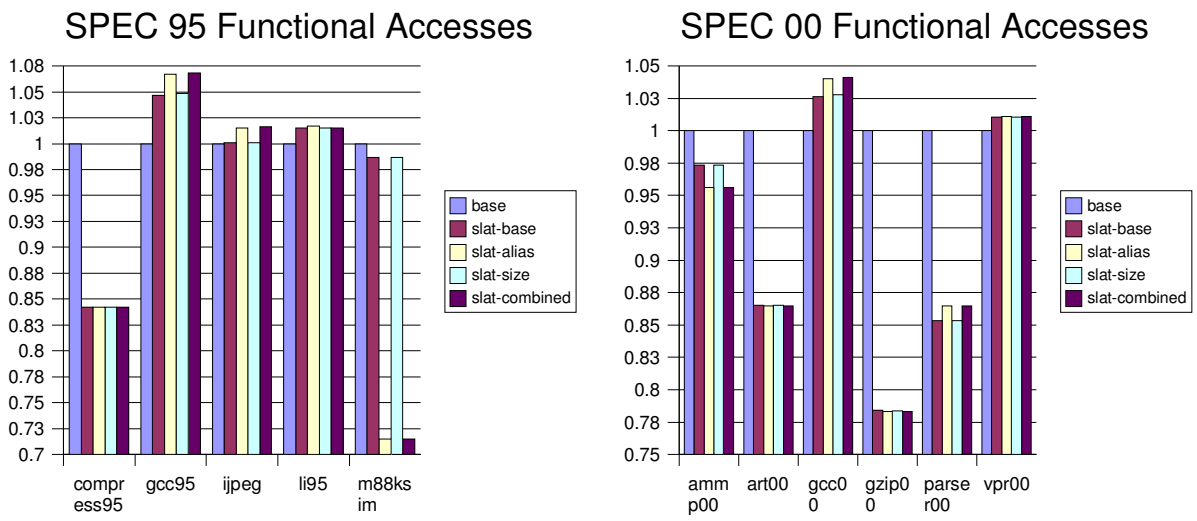


(a) SPEC 95



(b) SPEC 00

Figure 8.22: DL1 Ideal Combined Accesses



(a) SPEC 95

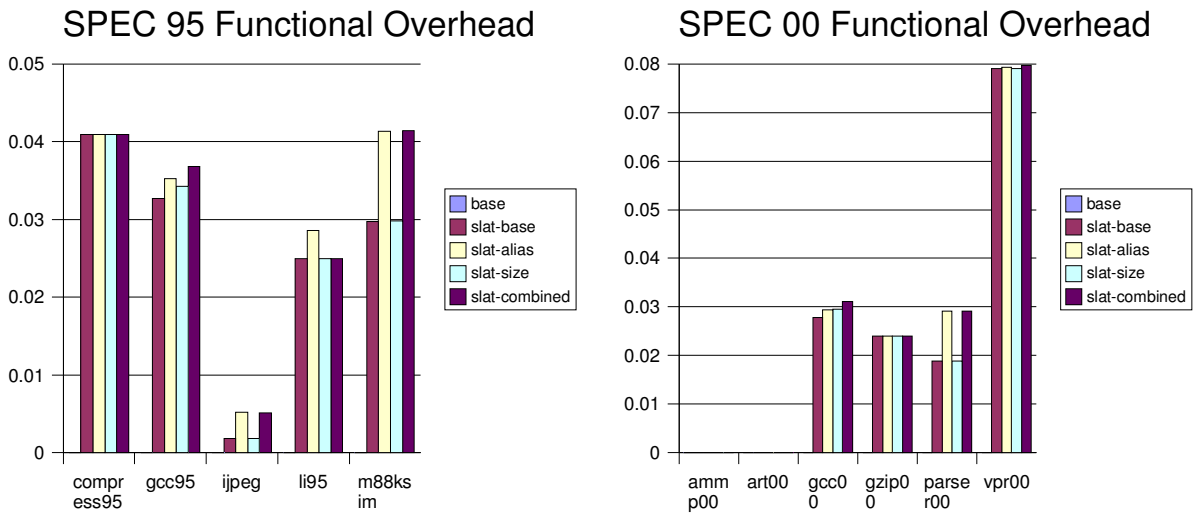
(b) SPEC 00

Figure 8.23: DL1 Functional Accesses

the extra hardware required.

8.6.4 Overhead

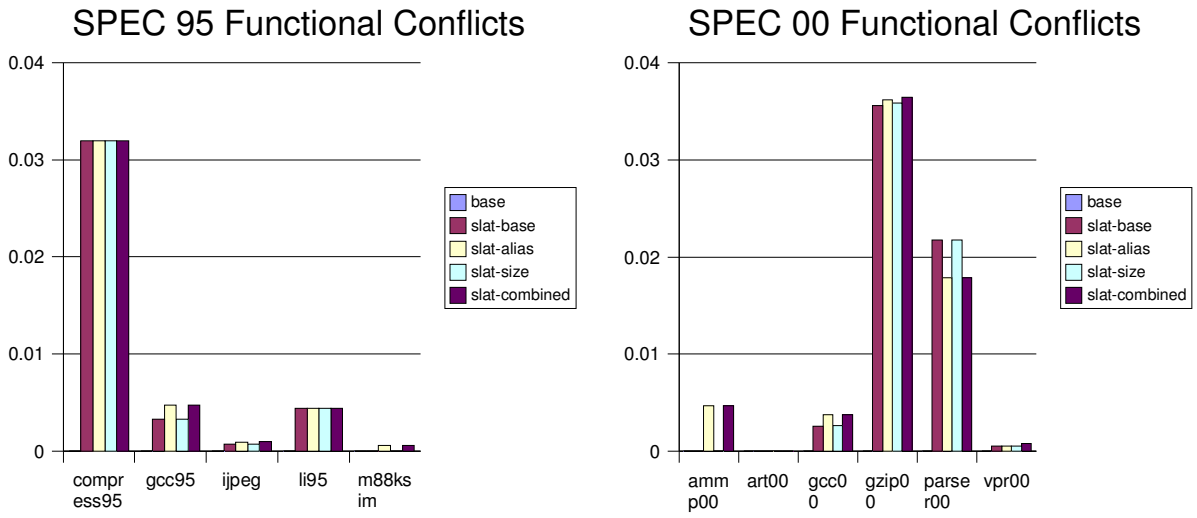
We were a bit disappointed by the performance and memory bandwidth results of the SLAT on some of the benchmarks. Our previous work did in fact indicate that some degradation was to be expected, but a 5%-6% degradation on some benchmarks is rather upsetting. After examining the experiment logs, we noticed something peculiar: many more `unmap` instructions were being executed than `map` instructions. We attribute this to branch prediction effects. While the `unmap` instructions appear after what should be an easily predicted branch (the loop back-edge), it is possible that other misspeculations could careen the processor off into code which incorrectly executes `unmap` instructions. It may also be possible that the predictor is *not* predicting the loop branches correctly due to aliasing in the predictor or some other reason. In any case, we decided to perform experiments to



(a) SPEC 95

(b) SPEC 00

Figure 8.24: DL1 Functional Overhead



(a) SPEC 95

(b) SPEC 00

Figure 8.25: DL1 Functional Conflicts

measure the inherent overhead cased by the SLAT.

For these experiments, we created a purely functional model of the SLAT. The processor simply executes instructions in sequence and maintains counts of all memory accesses and those accesses due to SLAT register save/restore overhead. Total accesses are presented in figure 8.23. In general, the cache bandwidth is either very similar or slightly lower than with the full timing model. The gzip benchmarks sees a slight improvement but most others hover around their corresponding timing results.

To better understand what is going on, we measured both the number of overhead memory operations due to saving and restoring SLAT entries when SLAT-mapped registers are saved and restored and the number of memory accesses that conflict in the SLAT and are thus eliminated serendipitously due to speculative promotion. For most benchmarks, the overhead outweighs the conflict benefit. This is to be expected since speculative promotion is performed because the compiler assumes the likely case that conflicts will not occur. It is interesting to note that conflicts and overhead very nearly cancel each other out on compress and parser, while gzip experiences a net gain due to the additional conflicts that convert memory operations to register copies.

8.7 Conclusion

Given the experimental results in this chapter, we conclude that size enhancements to the SLAT are not useful for these benchmarks. The alias enhancements enhance just one benchmark but do so in a quite dramatic fashion. We conclude that the LRT may be useful for other benchmarks but further study is necessary.

In particular, the SLAT and the LRT may be more useful in an object-oriented environment such as provided by the C++ language. This is because such languages tend to make

heavy use of pointers. C++ is particularly nasty in this regard due to its inherited pointer semantics from C. Class data members are essentially treated as global variables within the class code which makes them ideal candidates from speculative promotion. Furthermore, the emphasis on small procedures in object-oriented code means that less of the program is visible within a compilation unit which may further increase the utility of speculative promotion.

The LRT may be useful in other contexts as well. It is a general logical register renaming technique and is not tightly coupled to the SLAT. For example, the LRT could be used as an alternative mechanism to implement the register relocation architecture of Waldspurger and Weihl [119]. In fact it could be used to not only partition thread contexts as in the register relocation work but also to *combine* contexts so that, for example, unused simultaneous multithreading (SMT) contexts could be merged to provide a larger logical register file for threads that need it [120]. We believe the generalized program renaming and preprocessing provided by the LRT presents an interesting architecture for future research.

APPENDICES

APPENDIX A

MIRV Optimization Filters and Phase Ordering

Name	Description
alias	Alias analysis
arithSimplify	Arithmetic simplification/canonicalization
arrayToPointer	Reduce array accesses to pointer arithmetic
callGraph	Static call graph analysis
cleaner	Remove empty blocks
commAttr	Find unused variables and mark variables that are not register allocatable
CSE	Common subexpression elimination
deadCode	Dead code elimination
defUse	Reaching definition analysis
functCleaner	Remove unused functions
inline	Function in-lining
labelRemoval	Convert `goto` statements to a structured form
LICodeMotion	Loop invariant code motion
liveVariable	Live variable analysis
loopInduction	Induction variable strength reduction and test replacement
loopInversion	Loop inversion
loopUnroll	Loop unrolling
print	Print IR to a file
profile	Dynamic profiling (basic block, call graph and value)
propagation	Constant and copy propagation
reassociation	Expression reassociation/canonicalization
regPromote	Register promotion
replacement	Annotate IR with replacement attributes
scalReplAggr	Scalar replacement of aggregates
strengthReduction	Operator strength reduction

Table A.1: MIRV Front-end Filters

Name	Description
blockClean	Remove empty basic blocks
constant_propagation	Constant propagation
copy_propagation	Copy propagation
cse	Global common subexpression elimination
cselocal	Local common subexpression elimination
dead_code_elimination	Dead code elimination
dead_store_elimination	Dead store elimination
leafopt	Frame pointer removal
list_scheduler	Local instruction scheduling
list_scheduler_aggressive	Post-register allocation instruction scheduling
peephole0	Unnecessary cast removal
peephole1	Peephole optimization
post	Wait until after register allocation to run the remaining filters

Table A.2: MIRV Back-end Filters

Frontend Filters	Backend Filters
labelRemoval	peephole0
arrayToPointer={–fullReduction}	peephole1
loopInversion	blockClean
constantFold	cse
propagation	copy_propagation
reassociation	constant_propagation
constantFold	dead_code_elimination
arithSimplify	peephole0
deadCode	peephole1
loopInduction	cse
LICodeMotion	copy_propagation
CSE	constant_propagation
propagation	dead_code_elimination
CSE	peephole0
arithSimplify	peephole1
constantFold	list_scheduler
propagation	post
arithSimplify	list_scheduler_aggressive
constantFold	peephole0
strengthReduction	peephole1
arithSimplify	cselocal
propagation	copy_propagation
deadCode	dead_code_elimination
cleaner	peephole1
commAttr	blockClean
print	leafopt

Table A.3: Phase Ordering for O1 Optimization Level

Frontend Filters	**Backend Filters**
labelRemoval	peephole0
scalReplAggr	peephole1
loopUnroll	blockClean
arrayToPointer={–fullReduction}	cse
loopInversion	copy_propagation
constantFold	constant_propagation
propagation	dead_code_elimination
reassociation	peephole0
constantFold	peephole1
arithSimplify	cse
deadCode	copy_propagation
loopInduction	constant_propagation
LICodeMotion	dead_code_elimination
CSE	peephole0
propagation	peephole1
CSE	list_scheduler
arithSimplify	post
constantFold	list_scheduler_aggressive
propagation	peephole0
regPromote	peephole1
arithSimplify	cselocal
constantFold	copy_propagation
strengthReduction	dead_code_elimination
scalReplAggr	peephole1
arithSimplify	blockClean
propagation	leafopt
deadCode	
cleaner	
commAttr	
print	

Table A.4: Phase Ordering for O2 Optimization Level

Frontend Filters	**Backend Filters**
labelRemoval	peephole0
scalReplAggr	peephole1
callGraph={–topSort}	blockClean
functCleaner	cse
inline={–inlineSmallFuncs –inlineSingletons}	copy_propagation
loopUnroll	constant_propagation
arrayToPointer={–fullReduction}	dead_code_elimination
loopInversion	peephole0
constantFold	peephole1
propagation	cse
reassociation	copy_propagation
constantFold	constant_propagation
arithSimplify	dead_code_elimination
deadCode	peephole0
loopInduction	peephole1
LICodeMotion	list_scheduler
CSE	post
propagation	list_scheduler_aggressive
CSE	peephole0
arithSimplify	peephole1
constantFold	cselocal
propagation	copy_propagation
regPromote	dead_code_elimination
arithSimplify	peephole1
constantFold	blockClean
strengthReduction	leafopt
scalReplAggr	
arithSimplify	
propagation	
deadCode	
cleaner	
commAttr	
print	

Table A.5: Phase Ordering for O3 Optimization Level

APPENDIX B

Suggestions for Computer Architecture Researchers

In this appendix we reflect upon our research experiences in instruction prefetching as outlined in chapter 6. Given the presentation there, we make the following suggestions for improvement in computer architecture research practices:

- Research publications should be accompanied by a full statement of assumptions and/or source code for the software (simulators, compilers, etc.) used in the study.
- Funding should be made available for research groups to independently verify published work, as is done in other scientific fields.
- Such verification should be performed with a various sets of software tools such as simulators and compilers in order to increase the independence of the verification from the original work.

These suggestions are not a condemnation or judgment of any particular previously published work. It is not an attempt to discredit any individual or research group. Previous research was conducted under a set of practices and assumptions accepted at the time and we believe was published in as open and honest a manner as possible.

These suggestions are a set of guidelines for improving computer architecture research. We hope that some of these suggestions will make their way into regular practice and improve the quality of research. Though it is far from a complete list, we believe these suggestions will prove helpful.

It has long been recognized that the published work in the computer architecture field is difficult to reproduce. We allude to this in chapter 6 where points of ambiguity in instruction prefetching work are identified. The process of research verification in instruction prefetching has been a long one for us. Our experience has taught us that after some time has passed, researchers often don't recall their assumptions and no longer have the project setup to reference and obtain answers. Often these assumptions cover non-trivial implementations of the software architectural model and are impossible to reproduce without full information.

Our experience with Branch History Guided Prefetching (BHGP) is an excellent case study. At several points in the published paper, operation of the prefetching hardware is somewhat unclear. Only after obtaining source code for the simulator used in the study were we able to discover the flaws in our implementation and obtain results that verify the previous work. We applaud the researchers of that study for providing us the simulator code. We have found other groups less able to do so.

Given our experience, we suggest that publications include a full statement of all assumptions made in the study. We recognize that this is practically impossible as the software tools alone are much too complex to fully verify, much less obtain a list of all modeling parameters and algorithms used. The list would be much longer than the research study itself. Therefore, we propose that all published work should contain references to software source code that may be inspected and modified to verify the results obtained. In some cases, it is not possible to release source code due to commercial licenses. In such cases we

suggest that binary versions of the tools be made available so that other researchers may at least verify their own models against those used in the published work.

Currently it is difficult to obtain funding solely for verification of previous work, in contrast to common practice in other scientific fields. We believe that this is to the great detriment of the research community and industry as a whole. We hope that in the future such funds may be more readily available.

If verification studies are performed, we believe it is critical that they be performed with an independent set of software tools. Our studies in chapter 6 use MIRV and a new simulator tool, M5, not previously available to computer architecture researchers. The register allocation studies of chapters 7 and 8 used the same compiler as in earlier work but were conducted with a completely new simulator and software model of the proposed hardware. Use of these tools means that we have operated outside a baseline set of assumptions implied by the tool-sets used in the original work. This has helped us to identify the points of ambiguity in the previous work and has produced an interesting set of results for some of the previous work.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] S. C. Johnson, "Yacc: Yet another compiler compiler," in *UNIX Programmer's Manual*, vol. 2, pp. 353–387, New York, NY, USA: Holt, Rinehart, and Winston, 1979.

[3] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt, "Network-oriented full-system simulation using m5," in *Proceedings of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2003.

[4] M. A. Postiff, *Compiler and Microarchitecture Mechanisms for Exploiting Registers to Improve Memory Performance*. PhD thesis, The University of Michigan, 2001.

[5] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[6] Wall, "Global register allocation at link time," Tech. Rep. 86/3, Digital Equipment Corporation, Western Research Labs, 1986.

[7] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, pp. 127–145, 1968.

[8] K. Flautner, G. Tyson, and T. Mudge, "Mirvsim: A high level simulator integrated with the mirv compiler," 1998.

[9] G. A. Kildall, "A unified approach to global program optimization," in *Conference Record of the ACM Symposium on Principles of Programming Languages*, pp. 194–206, ACM SIGACT and SIGPLAN, ACM Press, 1973.

[10] M. Sharir, "Structural analysis: a new approach to flow analysis in optimizing compilers," *Computer Languages*, vol. 5, pp. 141–153, ???? 1980.

[11] B. K. Rosen, "High-level data flow analysis," *Communications of the ACM*, vol. 20, pp. 712–724, Oct. 1977.

[12] A. Ayers, S. de Jong, J. Peyton, and R. Schooler, "Scalable cross-module optimization," *ACM SIGPLAN Notices*, vol. 33, pp. 301–312, May 1998.

[13] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," in *Proceedings: 5th International Conference on Computer Languages*, pp. 229–240, IEEE Computer Society Press, 1994.

[14] G. Aigner and U. Hölzle, "Eliminating virtual function calls in C++ programs," in *ECOOP '96—Object-Oriented Programming* (P. Cointe, ed.), vol. 1098 of *Lecture Notes in Computer Science*, pp. 142–166, Springer, 1996.

[15] "C++ boost libraries." http://www.boost.org.

[16] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Reading (MA), USA: Addison-Wesley, 1996.

[17] H. Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley, 2000.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1995.

[19] *Intel(R) Architecture Software Developer's Manual*. Santa Clara, CA: Intel, 2000.

[20] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Upper Saddle River, NJ 07458, USA: Prentice-Hall, second ed., 1988.

[21] *Programming Language - C*. American National Standards Institute, 1990.

[22] S. C. Johnson, "Yacc—Yet Another Compiler Compiler," Technical Report CS-32, AT&T Bell Laboratories, Murray Hill , NJ , USA, 1975.

[23] B. Stroustrup, *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, 3 ed., 1997.

[24] *Programming Language - C++ ISO/IEC 14882:1998(E)*. American National Standards Institute, 1998.

[25] S. B. Lippman, *Inside The C++ Object Model*. Reading, Mass.: Addison-Wesley, 1 ed., 1996.

[26] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[27] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," *ACM SIGPLAN Notices*, vol. 30, pp. 1–12, June 1995.

[28] M. Lam, "Software pipelining: An effective scheduing technique for VLIW machines," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, (Atlanta, GA), June 1988.

[29] M. Smith, "Extending suif for machine-dependent optimizations," 1996.

[30] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale, Feb. 1985.

[31] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," in *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, 1981.

[32] M. W. Hall, *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston, Texas, USA, Apr. 1991.

[33] K. D. Cooper, M. W. Hall, and K. Kennedy, "A methodology for procedure cloning," *Computer Languages*, vol. 19, pp. 105–117, Apr. 1993.

[34] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, vol. 26,3 of *ACM Computer Architecture News*, (New York), pp. 227–237, ACM Press, June 27–July 1 1998.

[35] M. S. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. G. Abraham, "Achieving high levels of instruction-level parallelism with reduced hardware complexity," Tech. Rep. HPL-96-120, Hewlett Packard Laboratories, Feb.28 1996.

[36] D. A. Connors and W. mei W. Hwu, "Compiler-directed dynamic computation reuse: Rationale and initial results," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-99)*, (Los Alamitos), pp. 158–169, IEEE Computer Society, Nov. 30–Dec. 2 1999.

[37] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," *ACM SIGPLAN Notices*, vol. 29, pp. 183–193, Nov. 1994.

[38] *Intel(R) IA-64 Architecture Software Developer's Manual*. Santa Clara, CA: Intel, 2000.

[39] "Trimaran compiler toolset." http://www.trimaran.org.

[40] A.-R. Adl-Tabatabai, T. Gross, and G.-Y. Lueh, "Code reuse in an optimizing compiler," in *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pp. 51–68, 1996.

[41] S. W. K. Tjiang and J. L. Hennessy, "Sharlit—A tool for building optimizers," in *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 82–93, 1992.

[42] T. L. Veldhuizen, "Five compilation models for C++ templates," in *First Workshop on C++ Template Programming, Erfurt, Germany*, Oct. 10 2000.

[43] D. E. Knuth, *Semantics of Context-Free Languages*, vol. 2, pp. 127–145. New York: Springer-Verlag, June 1968.

[44] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan-Kaufmann, 1997.

[45] C.-K. Luk and T. C. Mowry, "Architectural and compiler support for effective instruction prefetching: a cooperative approach," *ACM Transactions on Computer Systems*, vol. 19, no. 1, pp. 71–109, 2001.

[46] T. J. Parr, *Language translation using PCCTS and C++: a reference guide*. San Jose, CA, USA: Automata Publishing Company, Jan. 1997.

[47] J. de Guzman, H. Kaiser, D. C. Nuffer, C. Uzdavinis, J. Westfahl, J. C. Arevalo-Baeza, and M. Wille, "Spirit v1.6.0." http://spirit.sourceforge.net.

[48] T. L. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, pp. 26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[49] M. A. Postiff, D. A. Greene, and T. N. Mudge, "The store-load address table and speculative register promotion," in *Proceedigs of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-33)*, (Los Alamitos, CA), pp. 235–244, IEEE Computer Society, Dec. 10–13 2000.

[50] D. R. Hanson, "Early experience with ASDL in lcc," *Software - Practice and Experience*, vol. 29, no. 5, pp. 417–435, 1999.

[51] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*. Redwood City, CA, USA: Benjamin/Cummings Pub. Co., 1995.

[52] American Telephone and Telegragh Company, *System V application binary interface: MIPS processor supplement: UNIX System V*. Upper Saddle River, NJ 07458, USA: Prentice-Hall, 1991.

[53] S. Chamberlain, R. Pesch, J. Johnston, and R. H. Support, *The Red Hat newlib C Library*. Red Hat, Inc., July 2002.

[54] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, 1991.

[55] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, Apr. 1997.

[56] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo, "TestTube: A system for selective regression testing," in *Proceedings of the 16th International Conference on Software Engineering* (B. Fadini, ed.), (Sorrento, Italy), pp. 211–222, IEEE Computer Society Press, May 1994.

[57] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy, "Generalized algorithmic debugging and testing," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 303–322, Dec. 1992.

[58] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *ACM SIGOPS Operating Systems Review*, vol. 27, pp. 203–216, December 1993.

[59] L. Barroso, K. Gharachorloo, and F. Bugnion, "Memory system characterization of commercial workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, vol. 26,3 of *ACM Computer Architecture News*, (New York), pp. 3–14, ACM Press, June 27–July 1 1998.

[60] J. E. Smith and W.-C. Hsu, "Prefetching in supercomputer instruction caches," in *Proceedings, Supercomputing '92: Minneapolis, Minnesota, November 16-20, 1992* (IEEE Computer Society. Technical Committee on Computer Architecture, ed.), (1109

Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. 588–597, IEEE Computer Society Press, 1992.

[61] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, vol. 25,2 of *Computer Architecture News*, (New York), pp. 252–263, ACM Press, June 2–4 1997.

[62] V. Srinivasan, E. Davidson, G. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 291–300, IEEE Computer Society Press, Jan. 2001.

[63] M. Annavaram, J. M. Patel, and E. S. Davidson, "Call graph prefetching for database applications," in *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 281–290, IEEE Computer Society Press, Jan. 2001.

[64] M. Annavaram, J. M. Patel, and E. S. Davidson, "Solving the instruction supply bottleneck in dbmss,"

[65] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec. 1978.

[66] C. Xia and J. Torrellas, "Instruction prefetching of systems codes with layout optimized for reduced cache misses," in *23rd Annual International Symposium on Computer Architecture (23rd ISCA'96), Computer Architecture News*, pp. 271–282, ACM SIGARCH, May 1996. Published as 23rd Annual International Symposium on Computer Architecture (23rd ISCA'96), Computer Architecture News, volume 24, number 5.

[67] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[68] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, (Paris), pp. 165–175, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 2–4, 1996.

[69] T. C. Mowry, "Private communication."

[70] T. C. Mowry, "Private communication."

[71] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, (Haifa, Israel), pp. 16–27, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Nov. 16–18, 1999.

[72] N. Jouppi, "Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, (Seattle, WA), May 1990.

[73] C.-K. Luk and T. C. Mowry, "Private communication."

[74] T. C. Mowry, "Private communication."

[75] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques," in *IEEE Transactions on Computers*, vol. C-48, pp. 1260–1281, 1999.

[76] A. J. Smith, "Sequential program prefetching in memory heirarchies," *IEEE Computer*, vol. 11, pp. 7–21, Dec. 1978.

[77] C.-K. Luk and T. C. Mowry, "Private communication."

[78] A. V. Veidenbaum, Q. Zhao, and A. Shameer, "Non-sequential instruction cache prefetching for multiple-issue processors," *International Journal of High Speed Computing (IJHSC)*, vol. 10, no. 1, pp. 115–??, 1999.

[79] G. Reinman, B. Calder, and T. Austin, "A scalable front-end architecture for fast instruction delivery," in *26th Annual International Symposium on Computer Architecture (26th ISCA'99), Computer Architecture News*, pp. 234–245, ACM SIGARCH, May 1999. Published as 26th Annual International Symposium on Computer Architecture (26th ISCA'99), Computer Architecture News, volume 27, number 2.

[80] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, (Paris, France), pp. 24–34, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 2–4, 1996.

[81] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative fetch and issue policies for the trace cache fetch mechanism," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, (Research Triangle Park, North Carolina), pp. 24–33, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 1–3, 1997.

[82] E. D. Vasanth Bala and anjeev Banerjia, "Transparent dynamic optimization: The design and implementation of dynamo," Tech. Rep. HPL-1999-78, HP Laboratories, June 1999.

[83] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, (Research Triangle Park, North Carolina), pp. 218–227, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 1–3, 1997.

[84] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin, "Classifying load and store instructions for memory renaming," in *Proceedings of the ACM International Conference on Supercomputing*, (Research Triangle Park, North Carolina), pp. 218–227, IEEE Computer Society TC-MICRO and ACM SIGMICRO, June 1–3, 1999.

[85] R. L. Sites, "How to use 1000 registers," in *Proceedings of 1st Caltech Conference on VLSI*, pp. 527–532, Caltech CS dept, 1979.

[86] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*. Los Altos, CA 94022, USA: Morgan Kaufmann Publishers, third ed., 2002.

[87] S. Vlaovic, E. S. Davidson, and G. S. Tyson, "Improving BTB performance in the presence of DLLs," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, (Monterey, California), pp. 77–86, IEEE Computer Society TC-MICRO and ACM SIGMICRO, Dec. 10–13, 2000.

[88] S. A. Mahlke, W. Y. Chen, P. P. Chang, and W. W. Hwu, "Scalar program performance on multiple-instruction-issur proceddors with a limited number of registers," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, vol. 1, (Kauai, HI), pp. 34–44, Jan. 1990.

[89] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings, Supercomputing '92: Minneapolis, Minnesota, November 16-20, 1992* (IEEE Computer Society. Technical Committee on Computer Architecture, ed.), (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. 808–817, IEEE Computer Society Press, 1992.

[90] D. G. Bradlee, S. J. Eggers, and R. R. Henry, "The effect on RISC performance of register set size and structure versus code generation strategy," in *The* 18^{th} *Annual International Symposium on Computer Architecture (ISCA)*, (Toronto), pp. 330–339, 1991.

[91] M. Postiff, D. Greene, and T. Mudge, "Exploiting large register files in general purpose code," Technical Report CSE-TR-434-00, University of Michigan, Department of Electrical Engineering and Computer Science, Apr. 2000.

[92] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, pp. 47–57, Jan. 1981.

[93] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105, ACM, ACM, 1982.

[94] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 501–536, Oct. 1990.

[95] P. Briggs, *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Apr. 1992.

[96] F. C. Chow, "Minimizing register usage penalty at procedure calls," in *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 85–94, ACM SIGPLAN, 1988.

[97] J. Lu and K. Cooper, "Register promotion in C programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, vol. 32, 5 of *ACM SIGPLAN Notices*, (New York), pp. 308–319, ACM Press, June 15–18 1997.

[98] A. V. S. Sastry and R. D. C. Ju, "A new algorithm for scalar register promotion based on SSA form," in *Proceedings of the ACM SIGPLAN'98 Conference on Programming*

Language Design and Implementation (PLDI), (Montreal, Canada), pp. 15–25, 17–19 June 1998.

[99] D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*. New York: McGraw-Hill, 1982.

[100] K. D. Cooper and T. J. Harvey, "Compiler-controlled memory," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), ACM SIGPLAN*, pp. 2–11, ACM SIGARCH v26/SIGOPS v32 n5/SIGPLAN v 33 n 11, Nov. 1998. Published as Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII), ACM SIGPLAN, volume 33, number 11.

[101] D. R. Ditzel and H. R. McLellan, "Register allocation for free: The c machine stack cache," in *Proceedings of the first international symposium on Architectural support for programming languages and operating systems*, pp. 48–56, ACM Press, 1982.

[102] H.-H. S. Lee, C. J. Newburn, M. Smelyanskiy, and G. S. Tyson, "Improving data cache architecture using region-based caching," in *IEEE Transactions on Computers (to appear)*.

[103] M. Postiff, D. Greene, S. Raasch, and T. Mudge, "Integrating superscalar processor components to implement register caching," in *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, (New York), pp. 348–357, ACM Press, June 17–21 2001.

[104] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 183–193, 1994.

[105] A. Klaiber, "The technology behind crusoe processors," technical report, Transmeta Corporation, Jan. 2000.

[106] M. J. Wing and T. C. Edmund J. Kelly, "Method and apparatus for aliasing memory data in an advanced microprocessor." United States Patent 5926832.

[107] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, (Los Alamitos), pp. 127–137, IEEE Computer Society, Nov. 30–Dec. 2 1998.

[108] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, (Los Alamitos), pp. 235–247, IEEE Computer Society, Dec. 1–3 1997.

[109] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchrinization of data dependences," in 24^{th} *Annual International Symposium on Computer Architecture*, pp. 181–193, 1997.

[110] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, Q1 2001.

[111] H. Dietz and C.-H. Chi, "CRegs: a new kind of memory for referencing arrays and pointers," in *Proceedings, Supercomputing '88: November 14–18, 1988, Orlando, Florida* (IEEE, ed.), vol. 1, (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), pp. 360–367, IEEE Computer Society Press, 1988.

[112] S. Nowakowski and M. T. O'Keefe, "A CRegs implementation study based on the MIPS-X RISC processor," in *International Conference on Computer Design, VLSI in Computers and Processors*, (Los Alamitos, Ca., USA), pp. 558–563, IEEE Computer Society Press, Oct. 1992.

[113] P. Dahl and M. O'Keefe, "Reducing memory traffic with CRegs," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, (San Jose, California), pp. 100–104, ACM SIGMICRO and IEEE Computer Society TC-MICRO, Nov. 30–Dec. 2, 1994.

[114] R. Yung and N. C. Wilhelm, "Caching processor general registers," in *International Conference on Computer Design*, (Los Alamitos, Ca., USA), pp. 307–312, IEEE Computer Society Press, Oct. 1995.

[115] A. González, J. González, and M. Valero, "Virtual-physical registers," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, (Las Vegas, Nevada), pp. 175–184, IEEE Computer Society TCCA, Jan. 31–Feb. 4, 1998.

[116] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (Vancouver, British Columbia), pp. 316–325, IEEE Computer Society and ACM SIGARCH, June 12–14, 2000.

[117] K. C. Yeager, "The MIPS R10000 superscalar microprocessor: Emphasizing concurrency and latency-hiding techniques to efficiently run large, real-world applications," *IEEE Micro*, vol. 16, pp. 28–40, Apr. 1996. Presented at Hot Chips VII, Stanford University, Stanford, California, August 1995.

[118] D. Sima, "The design space of register renaming techniques," *IEEE Micro*, vol. 20, pp. 70–83, Sept./Oct. 2000.

[119] C. A. Waldspurger and W. E. Weihl, "Register relocation: Flexible contexts for multithreading," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[120] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 392–403, ACM SIGARCH and IEEE Computer Society TCCA, June 22–24, 1995.