Compiler and Microarchitecture Mechanisms for Exploiting Registers to Improve Memory Performance

by

Matthew Allan Postiff

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
2001

Doctoral Committee:

Professor Trevor Mudge, Chair
Professor Richard Brown
Professor Edward Davidson
Assistant Professor Gary Tyson

Jesus saith unto him, “I am the way, the truth, and the life: no man cometh unto the Father, but by me.” – The Gospel of John, Chapter 14, Verse 6, The Bible.

Now unto Him that is able to keep you from falling, and to present you faultless before the presence of His glory with exceeding joy, To the only wise God our Saviour, be glory and majesty, dominion and power, both now and ever. Amen.

– The General Epistle of Jude 24, 25

The work herein is dedicated to this God.

Acknowledgments

Whenever acknowledgments are given, there is a danger of leaving someone out or forgetting something they have done. Hoping that I will not make any major omissions, I proceed to give thanks to those who have been close to me over the years, first in familial relationships, then in professional ones.

First of all, the living God of the Bible deserves much thanksgiving for his provision of the life, breath, and ability to use my mind which have been crucial to this work. Without Him, His sustenance and care, I could not even arise out of my bed every morning (Acts 17:28). O give thanks unto the LORD; for he is good: for his mercy endureth for ever (Psalm 136:1). He has enabled me to do this work.

Without question, my parents deserve the next thanks. Marvin and Vicki Postiff have been quiet and stable supporters over the last nearly 27 years. The number of things that they have provided for me are innumerable. In an age where the family is disintegrating before our eyes, they have maintained their marriage in a godly way, as it should be, for 28 years, and have been so helpful to me. Thanks Mom and Dad! My brothers Scott and Ben and my sister Deb have also been benefactors of this arrangement and my thanks to them also for being great to me. My grandmother Katherine Glenn was instrumental in my spiritual development early on; I thank her for being diligent in that regard. My grandmother Jacqueline Bianco has also shown a lot of interest and encouragement during this work.

So many friends have helped and prayed for me during this work. Steve Moss and Luman Strong, who have become good friends since October of 1997 when we began a Bible study together, have been constant supporters in prayer and other ways. My thanks to them. Bill Smith and his wife Andi have been close friends since 1999 when I first began attending Fellowship Bible Church, and my thanks are due them as well for their constant interest in my work and prayer support. Pastor Raymond Saxe and his wife Vivian have become close friends and spiritual mentors and I thank them for their encouragement and support. Thanks to the saints at Fellowship Bible Church as well for their support. Jerry and Nancy Benjamin have been close friends and mentors as well since my middle school days. Karl and Kelly Vollmar taught the small Sunday school class during my undergraduate studies. Thanks to both of these couples for their spiritual emphasis. Last, but certainly not least, is Naomi Kornilakis. She has been my closest friend over the last months and has been a constant encouragement. Thanks Naomi!

Beyond “family” support, Trevor Mudge, my advisor, has certainly been a major supporter over the past five years. Even before we worked together on this Ph.D., we had worked in a consulting arrangement with National Semiconductor Corporation to produce a 16-bit microcontroller design. Though we often have our disagreements in matters of faith, we have worked well together on technical subjects, for which I owe him thanks. The advisor/student relationship has been very easy because of his laid-back management style.

The person that I have worked most closely with the last few years has been David Greene. He has written the lion’s share of the code in the MIRV compiler, particularly in the frontend and optimization and analysis filters. He knows MIRV the best and was a constant source of information when I had difficulties with it. In addition, David provided the much needed sounding board for all of the ideas in this work. He has also been a co-author on all the latest papers I have written. Many thanks Dave!

This work would also not have been possible without the help of Charles Lefurgy, now at IBM. His work on MIRV has been invaluable. He worked on the function inliner, and his primary contribution, the MIRV high-level linker, has made this work much easier.

I would also like to thank Steve Raasch for his help on the register caching work which appears as a chapter in this dissertation.

Two old friends from my early school days are also due thanks. Mike Kelley and John Hall, now both at Tellabs, were lab partners during our undergraduate days and through the Master’s degree. Before that, we all went to the same middle school and high school. Thanks to Mike and John for their help on all of the class projects and other work that we shared. We worked a particularly long time together on the Microcontroller Memory Tester (fondly known as the MMT). This project “haunted” Mike and John while they were at the University, but it turned out that it actually worked!

Thanks also are due to Tim Strong who helped Mike and me build a multiply-accumulate circuit that was awarded in the University of Michigan VLSI Design Contest Experienced Class in 1998. An earlier design that Mike and I did, a CMOS floating point unit, received an honorable mention in the 1997 version of this contest. Tim also helped numerous times with computer software problems.

I would also like to thank Vijayalakshmi Srinivasan for her suggestions and encouragement throughout the course of this work.

My thanks are also due to David Helder, a former member of the MIRV development team, who wrote a number of the testing tools and backend optimizations as well as helped with debugging the compiler.

Thanks also to Kris Flautner, who originated the idea for the MIRV compiler and wrote several of the early works on the frontend analysis infrastructure.

My thanks are also due to the anonymous reviewers of several conferences for their comments, which significantly improved the quality of this research.

I would also like to thank those who provided the financial backing for my graduate studies. The people of the United States funded the first two years of my research through DARPA grant DAAH04-94-G-0327 (May 1996 to April 1998); they also funded the next two years on grant DABT63-97-C-0047 (May 1998 to April 2000). My thanks are also due to the University of Michigan Rackham Fellowship program, which funded my last year of research on a Rackham Predoctoral Fellowship. Simulations were performed on computers donated through the Intel Education 2000 Grant.

Thanks are also due to Microsoft Corporation and University of Michigan College of Engineering for providing generous scholarships to help pay for my undergraduate education, the stepping stone to this work. This made it easier on my parents, who paid the balance of my undergraduate education.

This project has been more than simply earning a degree from the University of Michigan. It has been accompanied with an effort to be a missionary for the sake of the Lord Jesus Christ. As His ambassador, I have made effort to show His gospel to those I with whom I have worked. While this is very difficult in a highly academic and intellectual environment, certainly those who have worked with me know my stand in the Christian faith. The major need of the day, which I have tried to communicate, is not for more computer engineering, but for people to understand their sinfulness, their need of someone to save them from an awful eternity in Hell, and the substitutionary work that the Lord Jesus Christ did in their behalf. All that is left is to believe on Him for salvation. To those who have heard this over and over, I thank you for being patient and listening. To those who have heard and not responded, I make request once more in writing that you hear the Word of God and turn to Him. Then, whatsoever ye do in word or deed, do all in the name of the Lord Jesus, giving thanks to God and the Father by Him (Colossians 3:17).

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

The performance of the memory hierarchy has become one of the most critical element in the performance of desktop, workstation and server computer systems. This is due in large part to the growing gap between memory and processor speed. Memory latency has been decreasing by 7% per year while processor frequency has been increasing at a rate of 55% per year since 1986 [Henn96]. This means that memory access time, as measured in processor clock cycles, is growing ever slower. Designers have found that using on-chip cache memory is one way to combat this *memory gap*. Putting caches on chip is possible because integration levels have been increasing according to Moore's law [Moor65]. The result is processor chips with 128KB or more of on-chip L1 and L2 cache [MDR99]. Still, the growing speed gap results in pipelines that are idle for as much as half of all cycles and CPIs no better than 1.5 on a two-issue superscalar processor [Barr98].

The memory gap is not the only problem with memory, however. The sheer number of load and store operations – 35% of instructions in the typical RISC instruction stream [Henn96] – place a great burden on instruction fetch, decode, dependence checking logic in addition to the cache and memory hierarchy. Binaries compiled to the CISC IA32 architecture have an even higher percentage of memory operations–50-80% in some cases [Vlao00].

The *memory bottleneck*, a term that encompasses both the memory gap and the frequency of memory operations in the system, forms the motivation for this research. This dissertation presents both hardware and software methods to implement and manage the register file more effectively. The register file is the highest level in the memory hierarchy and the more effectively it is used, the more the cache and memory hierarchy is shielded from memory operations. The overriding goal of this work is to eliminate memory instructions and their associated operations as much as possible.

The elimination of memory operations has several benefits. First, it reduces the amount of instruction fetch and decode that needs to be performed by the processor. Second, it reduces the pressure on memory load and store ports, freeing them for more critical memory operations and allowing the hardware to exploit more parallelism. Third, it directs operations to the higher-bandwidth register file, away from the lower bandwidth and memory system. Fourth, register operands make for easier dependence checking in both the hardware and the compiler. Fifth, register access latency is much lower than memory access latency–typically a register can be forwarded in zero effective cycles, whereas forwarding a memory operand to a dependent use usually incurs a one- or two-cycle pipeline bubble. All of these advantages can reduce power consumption as well.

This dissertation makes a number of contributions to the understanding and solution of the memory bottleneck. After considering background work in Chapter 2, the remainder of this dissertation is devoted to the study of the following:

1. Earlier studies of integer programs reported widely varying register requirements, some claiming that integer codes could use many registers and others claiming that only few were necessary. Chapter 3 shows, in contrast to some previous work, that a combination of pre-existing, aggressive compiler optimizations, when allowed to use a large number of registers, can indeed make use of those registers. The combination of the optimizations lead to performance improvements that are benchmark-dependent, non-additive, but significant across all the benchmarks studied. This chapter forms the basis and motivation for the remainder of the work by showing that register files larger than 32 registers are necessary and useful for the compiler.
2. Since a large register file is desirable, a mechanism is needed to implement one efficiently. The feasibility of a large register file is shown in Chapter 4, which introduces a new mechanism which integrates superscalar processor hardware and a novel renaming technique to implement a large architected register file using a caching mechanism. This approach is shown to be nearly as effective as an impossibly-fast large register file. Such a system makes it realistic to

implement a very large register file for use in systems that require either a flat file or a windowed file. The same technique can be used to implement a smaller rename register file for a cheaper microprocessor.

3. The larger register file has more capacity, which in some instances cannot be utilized because the conventional compiler is limited in its application of register allocation. This happens because of the compiler’s static uncertainty of address aliases. Chapter 5 introduces a new optimization called speculative register promotion and a new hardware structure called the store-load address table (SLAT) to address this concern. The SLAT can be used to promote scalars and array elements even in locations where it is not certain they can be legally allocated to a register using conventional means.
4. Once aggressive optimizations are performed, there is still a significant fraction of memory operations. Chapter 6 studies these operations, with a view toward understanding some hardware-software trade-offs and characteristics of the memory operations. Two new metrics are introduced for evaluation of the trade-off between compiler optimization and hardware design. The characteristics of problematic (frequently used or frequently missed) memory operations are shown as well.
5. Finally, Chapter 7 introduces a new research compiler called MIRV which was developed and used for this work. Various facets of the compilation framework and engineering are considered, including the intermediate representations used, the high-level source linker, the debug and test environment, and details of various filters and optimizations.

It is not the primary focus of this work to reduce effective memory access time through techniques such as prefetching or instruction scheduling, but instead to remove the memory operations altogether. Those techniques, as well as others to address cache miss penalties, are orthogonal to this work and can be used for additional performance benefits.

1.1. Theoretical Framework

The central processor of a computer has storage locations called registers that can be used to hold heavily-used scalar values or as a scratch pad for intermediate computational results. These registers can be accessed very quickly. Other data is stored in the main memory address space of the computer. This is commonly known as random access memory (RAM), which can be up to two orders of magnitude slower than a register. Thus, it is important to reduce the number of memory operations executed by the program and to increase the number of references that are made from registers. Cache memories are used as buffers between the processor registers and the main memory and reduce the impact of slow main memory speeds.

The compiler is a piece of software which translates human-written code in a programming language into computer-readable instructions. It is responsible for placing data into processor registers for fastest access. If it cannot allocate a datum into a register, then it must place it in RAM and direct the processor to use a memory operation to access the datum. Memory operations typically make up 30 to 35% of a program's total operation count. Somewhat surprisingly, this is true whether the computer is a reduced instruction set computer (RISC) or complex instruction set computer (CISC). So not only are memory operations expensive, but conventional compilers produce programs that use many of them.

1.1.1. The Source of Memory Operations

There are several reasons why memory operations make up such a large percentage of program execution. One may simply be that the computer's architecture does not provide enough registers. In processors that support multiple instruction issue (both superscalar and VLIW/EPIC styles) the availability of a large number of registers is particularly important. With only a few registers, program performance can be limited by spill instructions inserted by the compiler to deal with the small number of registers. These additional instructions can nullify the benefit of multiple issue as the extra issue slots are used for the data movement operations (overhead) instead of operations directly related to the algorithm.

A primary reason for the prevalence of memory operations is programmer-specified indirect memory accesses through arrays and pointers. The programmer uses indirection to solve his problem in an efficient way. Computed addresses for arrays, and stored addresses for lists, trees, and other structures are convenient and necessary to solve software problems. Such structures are often far too large to fit into any realistic register file.

Memory operations can also be attributed to compiler overhead. For example, many conventional compilers assign global variables to reside in memory for their whole lifetime and shuttle the variables into and out of the processor. Chapter 3 shows how allocating these variables to registers can significantly improve performance. Another example is function call overhead induced by the application binary interface. Variables which reside in registers are saved at the start of the function and restored at exit.

Aliasing is a another reason why memory operations are so prevalent. An alias is a condition where a datum is referenced through more than one name. The compiler cannot determine statically the values of *computed* addresses. Thus it often cannot be sure whether an alias could occur or not. It must also assume the worst–that the computed addresses can refer to the same location. The result is that the compiler must be conservative. It does this by leaving variables in memory and referring to them through load and store instructions. If it were to copy the value of a variable into a register and then the value of that variable was changed through a second name, the copy would be inconsistent with the actual value in memory. These extra load and store instructions are not always necessary, though, because the aliasing condition might not happen all the time. Chapter 5 shows how this problem can be addressed.

Dynamic memory allocation is another reason that data cannot be placed into processor registers. Dynamically allocated memory is also usually accessed with computed or stored addresses.

Most of these reasons have to do with the fact that registers can only be accessed with a direct address whereas memory can be accessed with a computed address. Those things which the programmer specifies through an indirect address generally cannot be allocated to registers.

Finally, computer input/output is specified with memory operations.

1.1.2. Alleviating the Memory Bottleneck

There are several approaches to deal with memory accesses. Perhaps the most important is to employ a small, fast “cache” memory structure as a buffer between the (fast) processor and the (slow) memory [Wilk65]. While numerous cache configurations have been proposed, caches cannot eliminate all of the accesses to memory. Primarily, this is because the cache must be small in order to be fast. Physical and cost limits prevent the cache from being as large as necessary to hold the entire working set of a program. Furthermore, caching relies on past information to predict what will be needed in the future. This means that caches cannot capture data at unanticipated addresses.

Prefetching of various forms, either in the compiler or in the hardware, have been utilized to address this problem, but this is still difficult due to the need for prefetching which is timely enough to avoid waiting for long memory latencies, and which is accurate enough to prefetch the right data and not pollute the cache [Vand00]. Data in Chapter 6 shows that the instructions which miss do so to many different addresses; this makes it difficult for a prefetching algorithm to determine the right address to prefetch.

Our approach is to attempt to allocate more data to the processor registers. This eliminates memory operations altogether. This is effectively a software caching technique.

1.2. Foundational Assumptions

This section enumerates two assumptions that are foundational to this work. First, we assume that binary compatibility is not of primary importance, unlike most previous work. Strict backward binary compatibility means that the architecture cannot be arbitrarily changed to accommodate new microarchitectural techniques. With the advent of binary-to-binary translation and optimization technology such as FX32!, Dynamo, and Transmeta’s code morphing software [Hook97, Bala99, Klai00], this constraint can be removed so the compiler, architecture, and system designer is free to select a better point in the design space than previously allowed. Still, many of the techniques proposed herein can be applied directly to existing architectures with little modification.

Second, we assume that the compiler is capable of complex analyses and transformations. Work in the early- to mid-1980s assumed it was too expensive to do global register allocation because of compiler runtime or software bugs [Harb82, Bere87]. It is evident from current compilers that complexity is not a concern as it was years ago–the IA64 architecture is evidence that complexity is shifting from the hardware to the compiler [IA6499]. We are not as concerned with compiler runtime as with the runtime of the generated code, especially in light of trends in processor speed and memory size that are evident in today’s processors. We do not leave the job completely up to the compiler, though, as is the case in several VLIW architectures. We believe the best trade-off is somewhere in the middle, where strengths are taken from both the compiler and hardware. This philosophy may mean some duplication of effort as some things may be done by hardware that were also done in the software.

Chapter 2
Background

This chapter on background work briefly outlines the various areas of research that are related to this dissertation. We start by discussing the basic trade-off between the use of registers and cache memory. We then look at basic register allocation and spilling and show how it fails to allocate variables to registers under conditions where aliases are present. Alias analysis, a compilation step which is necessary for correct optimization, is examined next. The optimization called register promotion is then described; it uses the results of alias analysis and tries to alleviate the aliasing problem somewhat as it moves scalars from memory into registers in regions where the compiler is sure there are no potential aliasing relationships. We then briefly compare work in compiler-based control and data speculation to this research. The chapter ends with some summary remarks.

2.1. Registers, Caches and Memory

A fundamental trade-off in computer architecture is the structure of registers and cache memory in the processor. This trade-off will be examined in this section.

The benefits of registers are primarily short access time and short instruction encoding. Registers are accessed with direct addresses which simplifies value lookup. Since there are generally few register locations (compared to memory locations), the register address can be encoded in a few bits. However, registers complicate code generation because machine calling conventions typically require some registers to be saved across function call boundaries. This is an important consideration since function calls occur frequently [Emer84]. There are many other trade-offs in the design of a register architecture for a processor. Table 2.1 catalogs them.

Registers	Cache
– storage size 1-128 registers (4B-512B)	+ storage size 256B-64KB typical
+ fast access (few and direct index)	– slower access (large, computed address, tags, memory management and protection checks necessary)
+ fewer address bits (less instruction bandwidth because of denser code)	– more address bits
+ lower memory traffic (fewer ld/st insts, cache and memory accesses, and power)	+ ld/st insts (expand the code again)
– more ld/st for synchronization at alias	+ no synchronization
– more ld/st at fcall boundary	+ no saves/restores at function call boundaries
– more ld/st at context switch boundary	– data can be automatically kicked out at context switch boundary
+ multiple ports less expensive because few entries	– more costly to multi-port because of many entries
+ easy dependence check for hazards	– hard dependence check
– aliases (computed addrs) and stale data	+ no need for aliases or stale data
– cannot take address of variable resident in (the C ’&’ operator)	+ can take address of variable resident in
– limited addressing modes (direct)	+ any addressing mode (computed)
– word-sized data only (ISA dependent)	+ any-sized data
– must have compiler to manage	+ dumb compiler will do

Table 2.1: Comparison of registers and cache.

Cache memory structures have also been studied extensively, at least back to 1965 with the slave memories described by Wilkes [Wilk65]. Since our focus is on the register file, caches will not be considered here in any further detail. An excellent early survey is [Smit82].

A primary concern of this dissertation is the problem that registers cannot generally contain aliased data. Because of the many positive aspects of registers, it is desirable that aliased scalar variables (and even non-scalars) should be referred to through register names. Caches and main memory can contain such aliased data because the hardware maintains consistency among the copies at the various levels. The movement of data from memory address space to the register file has in the past meant that consistency could not be maintained simply because address information was not associated with the register data. Registers are meant for extremely fast access and adding hardware to keep this consistency must not be allowed to slow them down too much.

The remainder of this chapter surveys some of the work related to register file design, register allocation, and alias analysis. The keys to understanding all of this previous research is that it attempts do one or both of the following:

1.Reduce the number of memory operations

2.Reduce the apparent latency of memory operations

Both are essential to microprocessor performance because of the growing gap between processor and memory speed [Henn96].

2.2. Register File Design

Research into the trade-offs between register files and caches has resulted in a wide variety of engineering solutions since the earliest days of computer architecture. Hardware registers (also called scratchpads) have been used since the early 1960s [Kuck78]. Table 2.2 gives a selected overview of this history. We will comment on a few of the machines listed in the table where they are relevant.

The CDC 6600 series machines had a total of 24 registers. Loads to 7 of the 8 address registers had interesting side effects. A load into register A[1,2,3,4,5] resulted in the data at that address being automatically loaded into data register X[1,2,3,4,5], respectively. Similarly, loading an address into A6 or A7 resulted in a store from X6 or X7 to that address. This allowed efficient encoding of vector operations because the load and store operations did not need to be explicitly specified. The Cray-1 later made this vector optimization explicit in its vector registers and instructions [Great].

The Cray-1 [Siew82] has a set of primary registers and a set of secondary or background registers. There are fewer primary registers, which allows them to be fast, while the secondary registers are slower but many in number. Long-lived values are stored in the secondary register files and promoted to the primary register files when used. The Cray-1 contains a total of 656 address and data registers (including the vector registers but not counting the control registers).

Machine	Description	Date, Refs
Burroughs B5000	Stack-based computer. Registers hold the top two values on the stack. Eliminates some data movement introduced by stack.	1961 [Lone61]
Ferranti ATLAS	128 24-bit registers for data or address computation; 1 accumulator register	1961 [Kilb62, Siew82]
ETL Mk-6	Still looking for info on this one...	1962 [Taka63]
CDC 6600	8 18-bit index regs, 8 18-bit address regs, and 8 60-bit floating point regs. Side effects of loading an address into an address register are described in the text.	1964 [Siew82, Great]
IBM System/360	16 32-bit integer regs, 16 64-bit floating point regs.	1964 [IBM98]
TI Advanced Scientific Computer (ASC)	16 base regs, 16 arithmetic, 8 index, 8 vector-parameter regs, all 32-bits	1966 [Siew82]
PDP-11	8 16-bit integer regs (PC and SP included), 6 64-bit floating point regs. Extended to 16 integer regs in 1972.	1970 [Siew82, DEC83]
Cray-1	8 24-bit addr (A) regs, 64 24-bit addr-save (B) regs, 8 64-bit scalar (S) regs, 64 64-bit scalar-save regs (T), 8 vector (V) regs. A vector is 64 64-bits regs.	1977 [Siew82]
VAX	16 32-bit regs for integer or floating point. (PC, SP, FP, and AP regs included).	1978 [Siew82, Brun91]
Intel 80x86, IA-32	8 integer, 8-entry floating point stack (16-bits, extended to 32-bits later)	1978 [Siew82]
Sparc	8 globals, 16-register window with 8 ins and 8 locals, as well as access to 8 outs which are the next window’s ins, all 32-bits. Number of windows from 3 to 32. 32 64-bit floating point regs.	1987 [Weav94]
AM29000	256 registers, all completely general purpose. 64 global, 128 “stack cache”, 64 reserved.	1987 [Great]
Alpha AXP	32 integer, 32 floating point, 64-bits each	1992 [Site92, Case92]
IA-64	128 integer, 64 predicate, 128 floating point registers, some with rotating semantics for software pipelining.	1998 [IA6499]

Table 2.2: A partial history of hardware registers.

The hierarchical register file, which is very similar to the Cray-1 organization, is proposed in [Swen88]. The authors present the classic argument that a large fast memory store can be simulated by a small fast store and a large slow store (in their case, 1024 registers). The results show speedups of 2X over a machine with only 8 registers. The tradeoffs noted include higher instruction bandwidth and storage, larger context switch times, and increased compiler complexity. The instruction bandwidth and storage requirement is reduced by including an indirect access mode where a short specifier can be used to indicate the source value “comes from the instruction which is N instructions before the current instruction.”

2.2.1. Large Architected Register Files

Sites presented perhaps the first in-depth discussion of the advantages of being able to support large numbers of registers, in his paper “How to use 1000 registers” [Site79]. He also noted the limitation caused by aliasing and coined the term *short-term memory* to denote a high-speed register set under compiler control. Besides cataloging some of the design issues related to short term memory systems, it was noted that it is often not possible to maintain data values in registers due to aliasing problems. If such values are placed in registers, they must be written and read from main memory as necessary to maintain coherence. Even though a machine may have 1000’s of registers, it is likely that most of them will be left unused by conventional compilers (in 1979).

2.2.2. No Architected Register File

Work done in the 1980s at Bell Laboratories took a different approach by suggesting the complete removal of registers from the compiler-visible architecture. The work was embodied in the “C-machine” and its “stack cache”, the “CRISP”, and later the “Habit” [Band87, Ditz87a, Ditz87b, Bere87a, Bere87b]. Instead of programmer-visible registers, the architecture has a “stack cache” which is a special purpose data cache for program stack locations; as such, it caches references to local scalar, array, and structure variables in the function linkage stack. The goal of this cache was to eliminate register allocation from the compiler and reduce the amount of data movement at function call boundaries. This allows the use of a large number of hardware registers without needing compiler allocation and without requiring every implementation of the ISA to have the same number of registers. Initial proposals were for 1024 registers in this cache, but the first reported implementation had 64 entries [Bere87a].

The essential features of the stack cache that distinguish it from a normal data cache are as follows: 1) it has no tags; 2) it caches a contiguous range of memory, i.e. the top of the program stack; and 3) the range being cached is delimited by high and low address registers. Alias checking is enabled by comparing any computed address with the high and low ranges of the stack cache; if the address falls within the limits of the high and low bounds, the data is in the stack cache. If not, memory is accessed (there is no other internal data cache in the processor). In this way, data objects can be allocated to the stack cache without fear of aliasing. Special handling is needed when the stack cache is overflowed, but this is rarely the case. The authors found that a large percentage of data addresses can be computed early in the pipeline because they are simple base+offset calculations where the base is the stack (or frame) pointer. The stack pointer remains constant for the life of the function (except when calling out to children functions).

The stack cache was designed to incorporate the best features of both registers and cache memory. It was direct mapped and had no tag comparison, so it was fast. The instruction encoding only required a short stack offset from the current stack frame, much like the short direct register specifier of a conventional architecture. The stack cache could hold strings, structures, and other odd-sized data. Finally, the compiler could take the address of a variable in the stack cache.

It is important to note that the C-machine research assumes that compilation is expensive and that compilers are hard to write correctly. Therefore, simplifying the compiler was the motivation for the decision to eliminate register allocation in favor of the more straightforward stack allocation of local variables. The single-pass compilers in the 1980s were not able to determine if a variable could be placed in a register because of aliasing. The requirement of simple compilers is no longer widely held, as evidenced by the large number of optimizing compilers for register-based architectures. In fact, later versions of the CRISP compiler used an optimization similar to register allocation to pack variables into the stack space in order to reduce stack cache misses. The other primary assumption in the CRISP work is that function calls are frequent and that overhead of the function linkage mechanism is very important to overall performance. This has been and is still true [Emer84, Ayers97] so an important criterion of a register-architecture is how it handles function calls.

2.2.3. Register Windows

The Sparc architecture's register windows [Weav94] are a hybrid register/memory architecture intended to optimize function calls. It is a cross between the C-machine's stack cache and a conventional single-level register file. Each subroutine gets a new window of registers, with some overlap between adjacent register windows for the passing of function arguments. Because the windowed register file is large and many ports are required to implement parallel instruction dispatch, Sun researchers proposed the register cache and scoreboard [Yung95a, Yung95b]. The register cache takes advantage of the locality of register reference and the fact that register file bandwidth is not utilized efficiently for large multi-ported files. This is another fundamental trade-off between registers and memory. Sun and others report that about 50% of data values are provided by the bypass network [Yung95a, Yung95b, Ahuj95] and there is an average of less than one read and 3/4 writes per instruction. The Sun work also noticed that a small number of the architected registers are heavily used (stack and frame pointer, outgoing arguments, etc.). Because of these factors, the register file cache can be quite small and still capture a large portion of the register references. Fully associative register caches of size 20 to 32 were found to have miss rates of less than a few percent. This can provide a significant savings in cycle time and power consumption compared to the 140-register file in the SPARC architecture (for an implementation with 8 windows [Weav94]). Other architectures that could benefit from a register cache include the IA-64 and AM29000 because they have a large number of architected registers. It is unclear from the previous work what the compiler could do to more evenly utilize the register file.

There is other work that attempts to reduce the implementation cost of large register files. One is a technique called "virtual-physical registers" which is described in [Gonz97, Gonz98, Monr99]. Here the goal is to allocate physical registers as late as possible so that their live ranges (in terms of processor cycles) are reduced. This is based on the observation that the actual lifetime of a value begins at the end of instruction execution rather than when the instruction is decoded. The difference could be a large number of cycles. Tags, called virtual-physical registers, are used to specify instruction dependencies, but these have no storage associated with them. The actual physical register storage is not allocated until instruction writeback. This has the effect of either 1) increasing the perceived instruction window size or 2) allowing the window to be reduced in size without negatively affecting performance. The second option is interesting because it allows the processor to implement a smaller number of physical registers. The only difficulty is that sometimes the processor may run out of physical registers and the instruction cannot be written back. In this case, the instruction is re-executed.

2.3. Register Allocation and Spilling

The problem of allocating scalar variables to registers, called the *register allocation problem*, is usually reduced to a graph coloring problem [Chai81, Chai82, Brig92, Chow84], where an optimal solution is well-known to be NP-complete. Other research has cast the problem as set of constraints passed to an integer programming solver [Good96, Kong98], or bin packing [Blic92]. We focus on graph coloring in this work because it is the most common technique for optimizing compilers. This section outlines some previous work in expanding the register set of an architecture so that the compiler can do more effective allocation and spilling.

Mahlke et. al. examined the trade-off between architected register file size and multiple instruction issue per cycle [Mahl92a]. They found that aggressive optimizations such as loop unrolling, and induction variable expansion are effective for machines with large, moderate, and even small register files, but that for small register files, the benefits are limited because of the excessive spill code introduced. Additional instruction issue slots can ameliorate this by effectively hiding spill code. This work noticed little speedup or reduction in memory traffic for register files larger than about 24 allocatable registers (often fewer registers were required)[1].

[1] We hypothesize that because of a conventional application binary interface [SysV91] and traditional alias management the compiler was not able to take advantage of any more registers.

Register Connection adds registers to the architecture. It does so in a way that is very careful to maintain backward compatibility and requires a minimum of changes to the instruction set architecture [Kiyo93]. Connect instructions map the logical register set onto a larger set of physical registers instead of actually moving data between the logical and physical registers. This is similar to register renaming [Toma67, Kell75] but is under compiler control so that register allocation and code optimization and scheduling can take advantage of the larger set of registers available. This technique is helpful for instruction sets with very few registers (8-16) but does not help much after 32 registers (where not much spill code is generated). The connection instructions were carefully designed to minimize execution delay and code size.

The compiler-controlled memory [Coop98a] combines hardware and software modifications to attempt to reduce the cost of spill code. The hardware mechanism proposed is a small compiler-controlled memory (CCM) that is used as a secondary register file for spill code. The compiler allocates spill locations in the CCM either by a post-pass allocator that runs after a standard graph-coloring allocator, or by an integrated allocator that runs with the spill code insertion part of the Chaitin-Briggs register allocator. A number of routines in SPEC95, SPEC89, and various numerical algorithms were found to require significant spill code, but rarely were more than 250 additional storage locations required to house the spilled variables. Potential performance improvements were on the order of 10-15% but did not include effects from larger traditional caches, write buffers, victim caches, or prefetching. These results show the potential benefit of providing a large number of architected registers–not only simplifying the compilation process in the common case, but also reducing spill code and memory traffic.

2.4. Alias analysis

Compiler alias analysis is yet another field related to the SRF. Alias analysis is important because it enables optimizations such as common sub-expression elimination, loop-invariant code motion, instruction scheduling and register allocation to be applied correctly to the program. While alias analysis is used to determine potential data dependencies for all of these optimizations, we view it as taking two distinct roles. The first is in register allocation, where it determines whether a variable can be *allocated* to a register or not. The second is in code transformation, where it determines whether a code *transformation* is legal. While both kinds of decisions are necessary for correctness (the overriding concern), the first is a data layout decision and the second is a code-layout decision. Alias analysis is used to ensure correctness of an optimization but if it is conservative it limits the scope and potential of applied optimizations. In other words, alias analysis is necessary, but *aggressive* alias analysis is needed to allow good optimization.

In deciding how the code-layout can be changed, the compiler is deciding whether it is semantically correct to move code out of loops, to eliminate redundant computations, or to otherwise re-arrange the code.

When the compiler addresses the data-layout problem, it must trade off the speed of the allocated memory against the functionality of it. In the case of an on-chip, direct addressed register file, the speed is very high but its functionality is low because data is accessed by statically specified indexes. Furthermore, the conventional register file does not have built-in checking for aliases between data in a register and data in memory.

The remainder of this section is organized into subsections describing the various previous research. These could also be divided into software, hardware, and combined hardware/software solutions.

2.4.1. Background on Alias Analysis

A location in a computer's memory is referred to by a numerical address which is computed during the execution of any instruction that accesses that particular location. Memory aliasing occurs when a storage location is referenced by two or more names. This can happen in languages like C that have pointers. Data at a memory location can be temporarily kept in a register only if we can assure that all instructions that might refer to that memory location can be made to refer to the register instead. Because instructions compute the address of the data they refer to at their time of execution, it is often impossible to tell before execution (i.e. at compile time) which instructions refer to a particular memory location; thus we run the danger of substituting two or more registers for what appears to be different memory locations, when we should have substituted only a single register. If this occurs, copies of the same data will be placed in two or more registers, leaving open the possibility that the copies can be changed separately. Thus data that was meant to represent the value of a unique variable can end up with two or more distinct values. Clearly this is wrong.

The allocation of data to registers is done by a compiler–the program that translates a programming language like C into basic machine instructions. The compiler analyzes a program before it executes and thus cannot detect if address aliasing does occur when the program runs. To avoid possible errors the compiler must make conservative assumptions about the values of addresses, and, as a consequence, must be conservative about what data can be kept in registers. This in turn means that whole classes of data cannot be placed in registers, at least for part of their lifetime.

Aliasing through memory is problematic because modification of a value through the use of one name will change the value examined through another name when both names refer to the same location in memory, (e.g., a[i] and a[j] may refer to the same location). If the compiler can determine with certainty that the names refer to disjoint locations, it is possible to allocate each name to a machine register where the value will reside. Similarly, if the compiler can be certain that both names always refer to the same location, it is possible to replace uses of both names with a single register name and allocate the location to a machine register.

Unfortunately, making such determinations is difficult. The use of pointers or accessing of arrays with different index variables creates new names. Furthermore, the pointer or index can be modified programmatically, thus changing the names at runtime. Such locations cannot be easily placed in registers because a traditional machine register has only one name.

Such values can be allocated to registers within regions of the program where the compiler can determine the exact set of names that refer to the location. In fact, this is a necessity in a load-store architecture, because the memory value must be placed in a register before use. However, such allocations are short-lived, because a modification of the memory value through another name will not be reflected by a change in the value in the register. Thus, the value must be updated by re-loading the value from memory. Likewise, any modification of the value through the register name must be written out to memory in case the value is accessed through an alias.

Languages with stronger typing than C allow the compiler to make more assumptions during alias analysis because only those names which have the same type as the variable can refer to the variable.

2.4.2. Modern Processor Memory Disambiguation

The conventional disambiguation hardware in a microprocessor (see for example Tyso97]) is not open to the compiler. This forces the compiler into the very conservative mode described in the last section, which requires loads and stores around the references to aliased data. Furthermore, loads cannot be moved past branches or stores on which they may (or may not) depend.

2.4.3. Static Analysis

The fact that aliasing information is not provided by the hardware to the compiled code forces most compilers to do static alias analysis to prove correctness of optimizations. Examples can be found in [Much97, Wils95, Rein98, Lu98, Emam94]. These references represent a range of complexity in the analysis phase; compile time is an important consideration in such analyses because they are so complex.

As far as register allocation is concerned, the simplest approach is to note which variables are potentially aliased and then simply not allocate them to registers. For code motion, simple heuristics can be employed to determine whether a load has a potential dependence on a previous store.

When the aliasing relationship between two instructions is not known, they can be moved relative to each other conditionally by the software. This is done by runtime memory disambiguation [Nico89], where explicit comparison instructions are used to route the code to the best execution path. If two addresses do not match, then the better code schedule can be selected. In the case they do match, the original, less aggressive code schedule must be provided.

2.4.4. Register Promotion

In function-level[2] register allocation, a variable is typically marked as ’allocatable’ or ’not allocatable’ depending on whether it can be resident in a register for its entire lifetime or not. In a conventional compiler and processor, a variable cannot be placed permanently into a register if there is more than one name used to access that variable. For the C language, if the address of the variable is taken the variable is said to be aliased and cannot be placed in a register. Global variables are also typically marked as “not allocatable” because register allocation algorithms are designed to run at the function level instead of the program level. Those variables which cannot be permanently allocated to registers are left in memory and require a load before each use and a store after each definition.

Register promotion [Chow90, Coop97, Sast98, Lo98] allows aliased variables to be placed into registers in code ranges where aliasing cannot occur. The variable is *promoted* to a register by loading it from memory at the beginning of the range. At the end of the range, the variable is *demoted* back to memory so that subsequent definitions and uses through other names are correctly maintained. It is apparent that this optimization increases register pressure because more values are maintained in registers during the non-aliased regions. The loads and stores are removed from these regions.

Several variants have been examined which use different code regions as the basic unit of promotion. In one, loops are considered as the basic range for promotion [Coop97]; another uses arbitrary program intervals as the promotion regions [Sast98]; and another did not consider explicit program regions but instead used a variant of partial redundancy elimination to remove unnecessary loads and stores [Lo98]. All of the previous work shows substantial reductions in the number of dynamic load instructions executed and varying reduction in the number of stores eliminated.

The promotion loads and demotion stores can be placed in infrequently executed paths in the control flow graph; this is shown to require more static loads and stores but results in fewer dynamic loads and stores. Register pressure was shown to increase by up to 25% in the most common cases [Sast98].

[2] “Function level” register allocation is typically called “global” register allocation. We use the former to avoid overloading the term “global.”

Explicit load/store instructions are needed for register promotion, and the compiler must demote a value to memory whenever there is a potential aliasing relationship.

2.4.5. CRegs

The Short-Term Memories described by Sites are the inspiration for CRegs [Diet88, Nowa92, Dahl94]. CRegs solves the aliasing problem much the same way as the Smart Short-Term Memory of this proposal. Registers have associated address tags which are checked against loads and stores to keep the registers and memory consistent. However, because the compiler may assign aliased variables to different registers, memory contents potentially have many duplicates in the CReg set. On a store, the associative lookup must find all copies of the data in the CReg array and update them accordingly. CRegs reduces the number of memory operations by eliminating redundant loads and stores from the program. These loads and stores were introduced in the conventional architecture because of aliasing.

2.4.6. EPIC

Work done at Illinois on the Impact EPIC architecture [Augu98] is concerned with scheduling load instructions ahead of control dependencies and aliased stores. Allowing loads to move past stores in the instruction schedule has a large impact on performance because otherwise the scheduling is very constrained. In previous work, the same researchers proposed the memory conflict buffer (MCB), which associates addresses with registers and tracks later writes to the addresses [Gall94]. In this way, a load can be scheduled above a store and the hardware will report when an aliasing condition actually occurs. Explicit check instructions which access the MCB state are required to determine if an aliased memory operation occurred between the check and the earlier, hoisted load instruction (called a *checked load*), at which time recovery code can be initiated. The recovery code and check instructions increase both the static and dynamic instruction counts, but the speedups reported are significant for benchmarks limited by memory ambiguities. The later research [Augu98] is concerned with generating efficient recovery code.

If a checked load instruction is followed by instructions which use the loaded value speculatively, exception information can be propagated through the uses so that one check can happen at the end of a long string of code. This reduces the number of check operations.

The Merced implementation of IA-64 utilizes a hardware structure very similar to the memory conflict buffer call the Advanced Load Address Table (ALAT). It allows the IA-64 compiler to advance load instructions and associated uses beyond branches and stores [IA6499]. To propagate exception information through a string of instructions, a NaT bit is employed. There is one per architected register. When the NaT bit is set on a register (say because of a page fault), all subsequent instructions which use that register essentially become NOPs and set their output register's NaT bit.

2.4.7. Memory Renaming

Tyson and Austin proposed memory renaming which allow loads to execute early in out-of-order processors [Tyso97]. This optimization is done entirely in hardware with no modification to the binary. This is achieved by tracking the loads and stores that frequently communicate with each other. Once a stable relationship has developed between a load and a store, the load's data can be accurately predicted to be coming from the store it is associated with. This allows memory to be bypassed entirely in the critical path–the address and the data are both predicted at once by the producer-consumer relationship between the load and store. A value file contains the data shared between the load and store. The key to early (speculative) resolution of the load is that the load and store PC's are used as the index of the value in the value file.

The prediction must be checked by performing the actual load, but this is off the critical execution path. The authors found that some of the load-store pairing comes from aliased data and global data, which they assume cannot be allocated to registers.

2.4.8. Other

Other work has focused on early generation of load addresses, prediction of load addresses, or prediction of load values in order to speed up program execution. None of these techniques assumed compiler involvement and thus worked with conventional binary programs.

2.5. Summary

This chapter has outlined a number of areas of research that are related to Smart Short Term Memories. Because the SRF relies on both hardware and software support, the previous work is a large body of computer architecture research. The previous work can be divided into three major categories: 1) that which deals with register file design; 2) that which deals with register allocation; and 3) that which deals with memory alias analysis and optimization in the face of aliasing. All of these are important foundational work to the designs considered in the rest of this dissertation.

Chapter 3
Register Utilization in Integer Codes

3.1. Introduction

Register allocation is an important optimization for high performance microprocessors but there is no consensus in the architecture or compiler communities as to the best number of registers to provide in an instruction set architecture. This chapter discusses reasons why this situation has occurred. Additional registers free compiler optimizations to be more effective, and the optimizations cause the compiler to need more registers. Essentially, if there are not enough registers and compiler optimizations to use them, the register requirements of programs will look artificially small. We show from a compiler perspective that, compared to the conventional 32-register file, 64 or more registers enables performance improvements from 5% to 20%. This is demonstrated with existing advanced compiler optimizations on the SPECint95 and SPEC2000 benchmarks. This work also documents that the optimizations eliminate cache hit operations, converting common-case cache hits to faster register accesses. Finally, this work provides additional measurements for the proper number of registers in a high-performance instruction set and shows that most programs can easily use 100 to 200 registers when multiple active functions are considered for simultaneous allocation to the register file. The performance improvements demonstrate that a large register file is important to high-performance computing, a result not clearly stated by previous research.

3.2. Background and Motivation

Large register files have many advantages. If used effectively they can: 1) reduce memory traffic by removing load and store operations; 2) improve performance by decreasing path length; and 3) decrease power consumption by eliminating operations to the memory hierarchy. Their merit derives from the fact that they require few address bits and act as a very high-speed cache.

On the other hand, large, multi-ported register files can become the critical path that sets the cycle time of the processor. There have been a number of proposals to circumvent this problem by implementing the register file either by physically splitting it, or providing a cache of the most frequently used registers and having a large backing store for the full architected set of registers. [Swen88, Yung95a, Yung95b, Gwen96]. The caching technique is particularly useful in a register renaming processor, where the architected (logical) register file can be much larger than the physical register file. In such a case, the physical register file acts as a cache for the larger architected register set. In this chapter, we assume that the implementation issues can be solved using such techniques.

Instead of considering implementation issues of large register files, we focus on their performance advantages. In scientific and DSP code, there are well-known techniques for exploiting large register files. However, a common view has been that large files are difficult to use effectively or are not necessary in general purpose code. In this chapter we will show that this is not indeed the case by applying several existing advanced compiler optimizations separately and then in combination. The results show that at least 64 registers are required for highest performance, and many more if the scope of register allocation is increased beyond the function boundary to include multiple functions which are active simultaneously on the call stack.

For control-intensive integer codes, the kind we are focusing on in this work, previous research to determine the best number of registers has not arrived at a clear consensus. For example, one study suggests that the number of processor registers that can be effectively used is limited to a couple dozen [Mahl92a]. Others have suggested that existing compiler technology *cannot* make effective use of a large number of registers [Beni93]. Studies on the RISC I architecture refer to earlier work which shows that 4 to 8 windows of 22 registers each (for a total of 80 to 144 registers) is sufficient to house the locals and parameters for over 95% of function calls [Tami83, Patt81]. In addition there has been much effort in the area of optimizing spill code; this is indicative of a need for more registers.

Floating point and DSP codes are generally thought to be able to take advantage of a large number of processor registers if the compiler utilizes advanced transformations such as loop unrolling, register renaming, accumulator and induction variable expansion, and software pipelining. One study confirmed this by showing that for a set of loop nests from the PERFECT and SPEC suites, register utilization increases by a factor of 2.6 after all ILP transformations were applied. While most optimized loops required fewer than 128 registers, the average register usage was about 70 for the loops studied [Mahl92b]. However, other work which used some similar benchmarks found that after 32 registers, adding registers produced only a marginal performance improvements, particularly if sophisticated code generation techniques are used [Brad91]. Finally another study showed that optimal performance for a number of Livermore kernels requires 128 to 256 registers [Swen88].

This lack of agreement in the architecture and compiler research communities is mirrored in commercial processor designs. Table 3.1 shows the register configuration of several embedded and workstation-class processors. While it is true that the register set size on some of these machines was constrained by backward compatibility and cost concerns, it is interesting to note the wide variety of register configurations. Incidentally, many of the high performance machines have larger physical register files to implement out-of-order execution. These additional registers are not available to the compiler.

There are several possible reasons for these mixed results, at least as far as integer codes are concerned. First, a small number of registers is suggested by the fact that a typical programmer does not usually have a large number of scalar variables in each function. Second, temporary variables are short-lived and can be packed into a small number of registers. Third, many data references are made through structure or array variables whose components cannot be easily allocated to registers. Fourth, variables local to a function are typically the only candidates considered for permanent register allocation. Though some global variables could be allocated to registers for their entire lifetime, most compilers leave them in memory and shuttle them in and out of registers with load and store operations. Fifth, local variables cannot generally live in the register file while a called function is executing unless a register windowing approach is used to house the variables from

Architecture Type	Architecture/Family	Register Configuration
Embedded	SHARC ADSP-2106x [AD96]	16 primary; 16 alternate; 40 bits wide
	TI TMS320C6x [TI97]	2 files of 16 registers; 32 bits wide; 1 Mbit on-chip program/cache/data memory
	Philips TriMedia TM1000 [TM1K]	128 registers; 32 bits each
	Siemens TriCore [Siem97]	16 address; 16 data; 32 bits wide
	Patriot PSC1000 [PTSC]	52 general purpose registers; 32 bits wide
Workstation	Intel IA32 [Inte96]	8 int; 8 float in a stack; 32 bits wide
	Transmeta Crusoe TM3120, TM5400 [Klai00]	64 int; 32-bits wide
	Intel IA64 [IA6499]	128 int, rotating and variable-size windows; 128 float; 64 bits wide; 64 1-bit predicates
	DEC Alpha 21x64 [Site92]	32 int; 32 float; 64 bits wide
	Sun Sparc V9 [Weav94]	8 globals, 16-register window with 8 ins and 8 locals, as well as access to 8 outs which are the next window's ins, all 32-bits. Number of fixed-sized windows from 3 to 32. 32 64-bit floating point regs.

Table 3.1: Register configurations for embedded and workstation processors.

multiple functions in different regions of the register file. Sixth, compiler optimizations are often deliberately throttled in order to avoid introduction of spill code. Thus 32 registers might appear to be sufficient because the optimizations have been artificially restricted to that level.

On the other hand, sophisticated compiler optimizations can significantly increase the number of variables by performing loop optimizations. These *optimization temporaries* can create greater register pressure than arithmetic and other simple temporaries since they live over longer ranges. The pool of register allocation candidates can also be enlarged by including array elements and scalar global variables. Furthermore, global variable register allocation can increase the number of registers that are required. Finally, variables that are aliased can be allocated to registers for some portions of their lifetime, further increasing the number of candidates for register allocation.

Many of the earlier studies did not have access to the advanced compiler transformations used today, either because the transformations were not known, or were not available in the compilers used for the studies. The lack of freely available optimizing compilers suitable for research studies has had a limiting effect on earlier research.

This chapter makes the case that for integer codes, a large architecturally-visible register file is necessary for high performance, and such large files can easily be utilized by current compiler technology. In fact, if the compiler does not use the right optimizations, the *apparent* register requirements of the program can be significantly lower than reality. To this end, we explore the performance effects of four sets of optimizations in this chapter. The first is traditional function-level optimizations. These increase register pressure by adding temporary scalar values and extending scalar lifetimes [Much97, Aho86]. The second optimization allocates global scalar variables and addresses to registers for their lifetime. The third is register promotion for aliased variables and global scalars. The fourth is function inlining, which effectively combines the live registers of two functions into one allocation pool. Each of these techniques increases the pool of register allocation candidates and depending on the benchmark can significantly impact performance. All of these optimizations have been examined in previous work but this study examines them in combination in order to present the register requirements of a modern optimizing compiler.

The presentation will proceed as follows. Section 3.3 describes the compilation environment and includes descriptions of the optimizations that we report on in this chapter. Section 3.4 describes our simulation setup. In Section 3.5 we present our experimental results. Each transformation is analyzed for its effects on overall performance, register pressure, and other effects. In addition, an incremental analysis is provided to examine how much benefit the non-traditional transformations provide over classical optimization. Section 3.6 presents some more evidence that a large number of registers is necessary for integer codes. Section 3.8 describes some previous work on register architecture. We summarize our conclusions in Section 3.9 and provide directions for future research.

3.3. Compilation Environment

This section describes the optimizations that were applied to increase register usage. We use the MIRV C compiler. MIRV is an ANSI C compiler which targets the Intel IA32 and SimpleScalar PISA architectures; for this work, the PISA architecture is used because it offers the ability to use up to 256 registers. The baseline optimization level is -

O1, which includes classical optimizations. The compiler performs graph-coloring register allocation and does scheduling both before and after allocation. The number of registers available for allocation is passed as a command-line argument to the compiler. A complete description of the MIRV compiler, the types and order of optimizations, and a comparison against GCC is outside of the scope of this chapter. This information can be found in our technical report [Post00a].

We use MIRV to transform the C files of the SPECint95 and SPEC2000 benchmarks into the MIRV high-level intermediate representation (IR). The IR files for each module are then linked together, producing a single “linked MIRV” file for the benchmark. Once linking is done, the MIRV compiler can apply the optimizations that are used in this work to show that general purpose code can take advantage of a larger register set. The following subsections briefly describe these optimizations.

3.3.1. Register Promotion

Register promotion allows scalar values to be allocated to registers for regions of their lifetime where the compiler can prove that there are no aliases for the value. The value is *promoted* to a register for that region by a load instruction at the top of the region. When the region is finished, the value is *demoted* back to memory. The region can be either a loop or a function body in our compiler. The benefit is that the value is loaded once at the start of the region and stored once at the end, and all other accesses to it during the region come from a register allocated to the value by the compiler.

Constant values that cannot fit into an instruction can also be promoted to a register in order to eliminate load-immediate instructions. Finally, indirect pointer references can also be promoted to a register. The register promoter in MIRV can promote global scalar variables, aliased local scalar variables, constants, and indirect pointer references (we call these *dereferences*). For the remainder of this chapter, we will only consider loop regions.

```
/* C source code */     # Assembly
                        .sdata
int *ptr = &pointedTo;      .globl ptr
                        ptr:
                            .long myAddrTaken

                        ...
...
                        lw $2,ptr
i = ptr;

                        # Disassembled object code
                        lw $2,-32760($28)
```

Figure 3.1: The conventional way that global variables are referenced.

3.3.2. Link-time Global Variable Register Allocation

One shortcoming of register promotion is that promoted global scalars are moved into and out of registers each time the loop (promotion region) is invoked. This causes unnecessary load and store instructions, as shown in Figure 3.1. In this example, a global variable called `ptr` is initialized to point to another variable. The assembly code on the right shows that this results in a load operation. This load operation is a simple load relative to the global pointer (`$28`). On other machines, it might be even more expensive to reference a global variable. MIRV can allocate global variables to registers for their entire lifetime, avoiding all such overhead. This eliminates the need to move the globals in and out of registers at region boundaries.

Post-link allocation can be used separately from register promotion, in which case it achieves much the same effect, in addition to avoiding the movement of values at region boundaries. It could also be applied after register promotion. In our experiments, we consider its application apart from promotion to isolate its benefit.

Allocating Global Variables

Instead of allocating globals into the set of local registers, we allocate them to a separate area of the register file. This is similar to the global registers in the SPARC and IA-64 architectures [Weav94, IA6499]. After linking the program's intermediate representation into one file and doing the appropriate coalescing of global variable declarations and definitions, the compiler determines if the global's address is ever taken and annotates that into the global's declaration in the intermediate representation.

Next we run the MIRV SimpleScalar/PISA backend and select the global variables to allocate to registers. Because the usage of registers 0-31 is fixed by the System V ABI, MIRV allocates global variables to registers 32..32+N, where N is set by a compiler command-line option. A global variable must satisfy two conditions before it is considered a candidate for permanent allocation to a register. First, its address cannot be taken anywhere in the program. This ensures that we can allocate the variable to a register for its entire lifetime without being concerned about aliasing conditions. Second, the global cannot be an imported variable. The second condition ensures that we only allocate user variables to registers—we have opted not to modify the C library code and thus do not attempt to allocate imported library variables to registers.

MIRV uses three heuristics to determine which global scalars to put into registers. The first is a simple first-come-first served algorithm. It selects the first N allocatable global variables and allocates them. The second heuristic is based on a static frequency count estimate. It is determined by the following formula:

$$\text{static frequency count} = \sum_{\substack{\text{all uses and}\\ \text{definitions}}} (\text{loop nest level ? } 4 \ll \text{loop nest level : } 1) \qquad \textit{(Eq. 1)}$$

That is, a reference to a variable within the top level of a function is given weight 1. A use within a loop is given weight 4 << 1 = 8. A use within a doubly-nested loop is given weight 16, and so on. Candidates are sorted by frequency count with the highest frequency variables getting preference for registers.

The third and final heuristic is based on a dynamic frequency-of-use. The program is instrumented with counters which are used to keep track of the execution frequency of the block statements in the program. The program is then run on a training data set and the final counter values are used to compute how many times a given global variable was accessed. This data is more accurate than the static frequency count estimate.

We only consider the static frequency count heuristic in this chapter since the first-come-first-serve is dependent on variable declaration order and the dynamic heuristic does not provide appreciable performance improvements when there are more than a few registers available for globals. We call the configurations based on the static heuristic “statX” where X is the number of registers in the machine. In our experiments, X minus 32 of those registers are permanently set aside for global variables; the original 32 registers are left to behave according to the ABI specification.

Once the candidates are chosen, they are allocated to a register by the MIRV backend. If the variable being allocated has an initial value, initialization code is inserted into the beginning of `main()`.

We do not use lifetime ranges to compact more global variables into the given number of registers. We think that trying to compact globals into fewer registers will not yield much appreciable benefit unless the machine has very few registers. Of course, if an existing ISA is used, then there are very few registers in general so it would be beneficial to try to compact them. Compacting globals by sharing registers requires movement of data between memory and registers. Register promotion does exactly that, so it is evident that these two optimizations are related.

Allocating Global Constants

Global arrays are responsible for a large portion of the computation in some programs. Each time a global array is referenced, its base address is formed and then used as part of an indexing calculation. This address formation (or materialization) takes two instructions on the PISA (MIPS IV) ISA because of the range limit on immediate constants. Even with loop invariant code motion that moves these address computations out of loops, many such instructions are executed.

We optimize this case by creating a new global scalar variable pointer which is initialized to point to the base of the global array. This pointer is then used in all array indexing computations instead of materializing the address. The new pointer variable is allocated to a global register for the entire program so that instead of materializing it at each point of use, a single register reference is all that is required. This reduces the number of load-immediate and add instructions that are executed. We call this optimization *address promotion*. Address promotion should not be performed if the pointer cannot be allocated to a register because rematerializing the address is probably cheaper than loading it from memory.

Loop invariant code motion can move many of these instructions out of local loops because they compute an invariant base address. Promoting the base address into a register removes all such instructions entirely instead of just moving them outside of the loops, which is important for functions which are themselves called within a loop. This optimization may seem wasteful at first because it is allocating a constant value (label) to a register. However, the frequency of these operations and the ease with which register allocation can memoize them make these operations prime candidates for optimization. Thus address promotion eliminates redundant computation by memoization. This effect is present in the go benchmark as will be mentioned later.

Local arrays cannot benefit from this optimization since their base address is already essentially in a register (at a constant offset relative to the frame pointer).

3.3.3. Inlining

MIRV inlines two kinds of functions: those that are called from one call site and those that are "small." Functions that are invoked from one call site are called *singletons*. There are a surprising number of singleton functions in the SPEC benchmarks as shown in the fifth numeric column of Table 3.2. From 17% to 63% of functions are called from a single call site. These can be inlined profitably without causing code expansion because the original function can be removed after inlining.

Singleton functions are inlined in MIRV if the estimated register pressure is below a certain bound. Otherwise, if inlining were performed, there might be extra spill code introduced. For example, if there are 3 live registers at the call site and a maximum of 8 live registers in the singleton callee, then inlining will require 11 registers whereas before inlining, 8 registers was sufficient because of the prologue spills and epilogue reloads present in the singleton callee. As more registers are made available to the inliner, this quickly becomes a non-problem.[1] The bound is set to the total number of registers available to the compiler for the specific simulation (either 32, 64, or 256 in this chapter).

[1] Some have disagreed with our approach. It is true that there are no more live *values* before inlining than after. However, combining the allocation candidates of the two functions together can cause spill code which performs worse than the original prologue/epilogue code. Our heuristic for this is to look at register pressure to determine if more spilling code will be introduced.

Small functions are those with fewer than 10 C statements. No register pressure measurements are examined for small function inlining.

Category	Benchmark	Total Functions	% Functions called from 0 or 1 call sites	
SPECint95	compress	30	23%	40%
	gcc	2,046	15%	26%
	go	383	0%	63%
	ijpeg	475	46%	29%
	li	382	49%	17%
	m88ksim	291	22%	30%
	perl	349	4%	26%
	vortex	959	29%	33%
SPECfp2000	ammp	206	23%	42%
	art	41	12%	34%
	equake	36	11%	53%
	mesa	1127	56%	25%
SPECint2000	bzip	92	16%	34%
	gcc	2275	14%	27%
	gzip	140	16%	31%
	mcf	39	8%	54%
	parser	349	6%	36%
	vortex	959	29%	33%
	vpr	301	5%	43%

Table 3.2: Functions in SPEC called from 0 or 1 call sites (-O1 optimization).
These are compile-time measurements and do not include functions never called dynamically.

Small functions are currently inlined until the code is expanded to a user-defined limit. The limit we use for this chapter is 10%, which means that we do not allow inlining to expand the static code size by more than 10%. This is not a problem in practice. For example, inlining all the small functions in *go* at every call site only results in a 7% code expansion. Once a singleton is determined to be suitable for inlining based on our register pressure heuristic, the function is inlined without regard for code expansion because the original copy of the function will be deleted after inlining. The net code expansion will be *about* zero.

A potential negative effect of inlining is that of poor code layout. Inlining a function can make performance worse if functions are aligned in such a way that they conflict with each other or if the working set size of the program has been made large enough that it does not fit into the cache. These issues are not examined in this chapter but are described elsewhere in the literature [Chen91].

3.4. Simulation Environment

All simulations were done using the SimpleScalar 3.0 simulation toolset [Burg97]. We have modified the toolset (simulators, assembler, and disassembler) to support up to 256 registers. Registers 0-31 are used as defined in the MIPS System V ABI [SysV91]. Registers 32-255 are used either as additional registers for global variables OR additional registers for local caller/callee save variables.

We ran variants of the SPEC training inputs in order to keep simulation time reasonable. Table 3.3 shows the data sets used for these experiments. Our baseline timing simulator is the default `sim-outorder` configuration; it is detailed in Table 3.4. Detailed information is available elsewhere [Post00a].

Category	Benchmark	Input
SPECint95	`compress`	30000 q 2131
	`gcc`	regclass.i
	`go`	9 9 null.in
	`ijpeg`	specmun.ppm, -compression.quality 25, other args as in training run
	`li`	boyer.lsp (reference input)
	`m88ksim`	ctl.lit (train input)
	`perl`	jumble.pl < jumble.in, dictionary up to ’angeline’ only
	`vortex`	250 parts and 1000 people, other variables scaled accordingly
SPECfp2000	`ammp`	test input modified so that numstp is 1, short.tether has the first 162 lines, and all.init.ammp has a subset of the statements in the original
	`art`	-scanfile c756hel.in -trainfile1 a10.img -stride 2 -startx 134 -starty 220 -endx 139 -endy 225 -objects 1 (test input)
	`equake`	< inp.in (test input)
SPECint2000	`gcc`	first 1442 lines of test input cccp.i
	`gzip`	input.compressed 1 (test input)
	`mcf`	inp.in (test input)
	`parser`	2.1.dict -batch < test.in
	`vortex`	250 parts and 1000 people, other variables scaled accordingly
	`vpr`	net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8 (test input)

Table 3.3: Description of benchmark inputs.

SimpleScalar parameter	Value
fetch queue size	16
fetch speed	1
decode, width	4
issue width	4 out-of-order, wrong-path issue included
commit width	4
RUU (window) size	64
LSQ	32
FUs	alu:4, mult:2, memport:2, fpalu:4, fpmult:1
branch prediction	2048-entry bimod, 4-way 512-set BTB, 3 cycle extra mispredict latency, non-spec update, 8-entry RAS
L1 D-cache	128-set, 4-way, 32-byte lines, LRU, 1-cycle hit, 16KB
L1 I-cache	512-set, direct-mapped 32-byte line, LRU, 1-cycle hit, 16KB
L2 unified cache	1024-set, 4-way, 64-byte line, 6-cycle hit, 256KB
memory latency	60 cycles for first chunk, 10 thereafter
memory width	8 bytes
Instruction TLB	16-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty
Data TLB	32-way, 4096 byte page, 4-way, LRU, 30 cycle miss penalty

Table 3.4: Simulation parameters for sim-outorder (the defaults)

3.5. Experiment Results

The results of our experiments are presented in the following subsections. The baseline in all cases is MIRV -O1 with 32 registers. In each set of graphs in the following subsections, the first two bars are the MIRV -O1 results with 64, and 256 registers. The graphs show simulated cycles for each of the SPECint and SPEC2000 benchmarks described in the previous section. Each figure shows the performance improvement of the respective optimization compared to -O1 with 32 registers so that it is clear what the individual optimization is doing to performance. Section 3.5.1 shows -O1 versus -O1 with register promotion. Section shows -O1 versus -O1 with link-time allocation. Section 3.5.2 shows -O1 versus -O1 with inlining. Section 3.5.3 shows the combination of all 4 sets of optimizations. The final subsection, Section 3.5.4, shows some information on the cache effects of the optimizations.

Note that -O1 performance can actually become worse when the number of local registers is increased because of extra spill code in the prologue and epilogue to handle callee save registers.

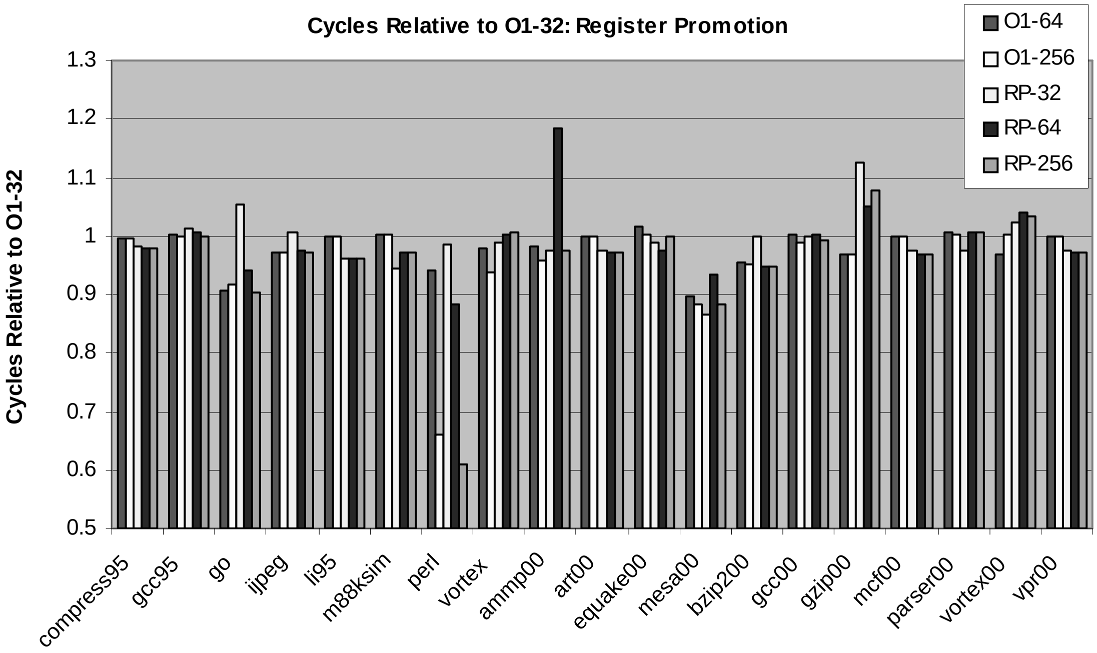


Figure 3.2: Performance of register promotion.

3.5.1. Register Promotion

Our first set of results, Figure 3.2, shows the performance increases due to register promotion. In addition to the two -O1 bars, each graph has three bars for register promotion, for 32, 64, and 256 registers. Register promotion improves performance from 5% to 15% on some benchmarks. Other benchmarks, most notably `li`, actually get worse with register promotion. This is due to spilling caused by the promotion. Figure 3.3 and Figure 3.4 show the load and store instruction reductions from these optimizations. A few of the benchmarks have 10% reductions in loads and stores; the notable exception is `art`, where there is nearly 50% reduction in load instructions due to promotion.

Examination of `art`'s benchmark source reveals five integer global variables which are used heavily in loops for comparison and array indexing operations. In addition, there are several global pointers into arrays and other data structures that are promoted. Promotion provides a significant performance win by reducing the number of loads by 46%. A small reduction in store instructions (2.3%) indicates that these variables are not

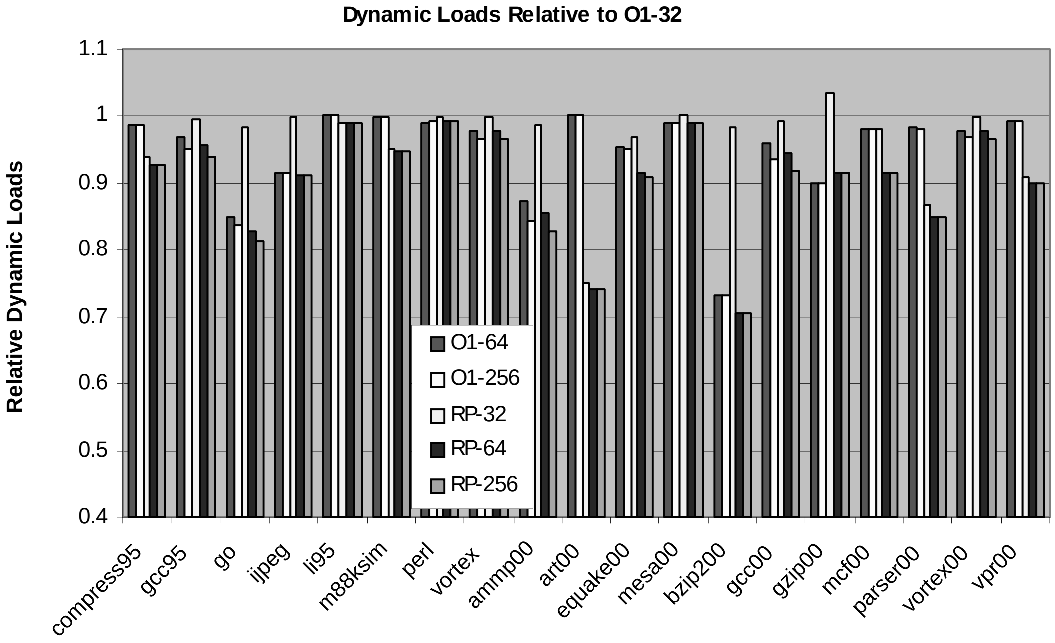


Figure 3.3: Loads eliminated by register promotion.

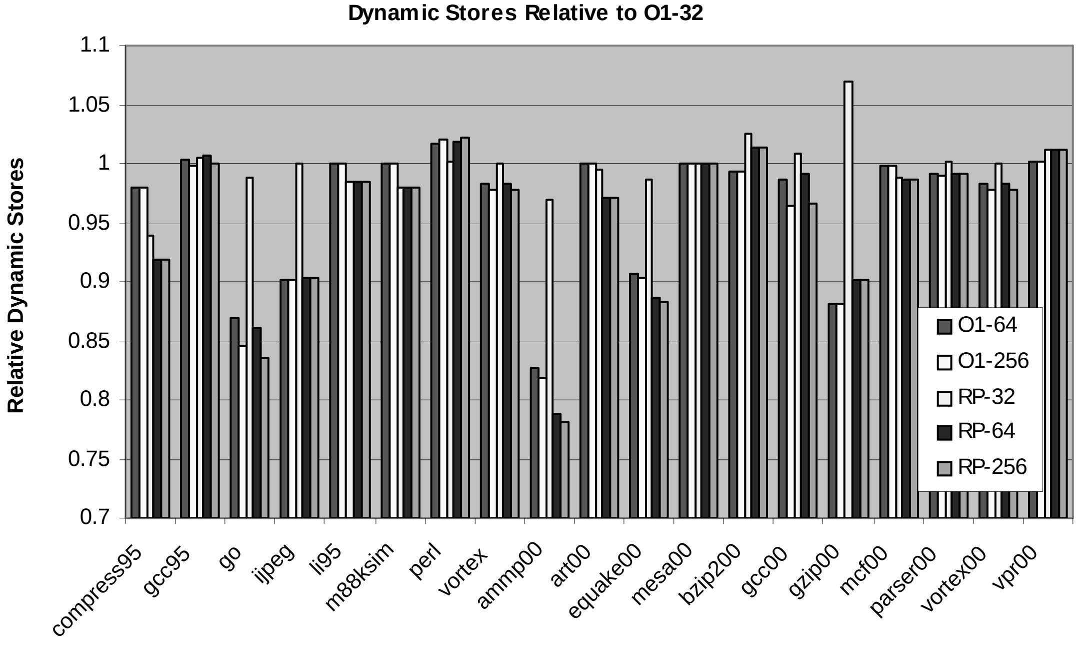


Figure 3.4: Stores eliminated by register promotion.

modified often by the program. These variables may not require full promotion but can use a weaker form that only considers load instructions.

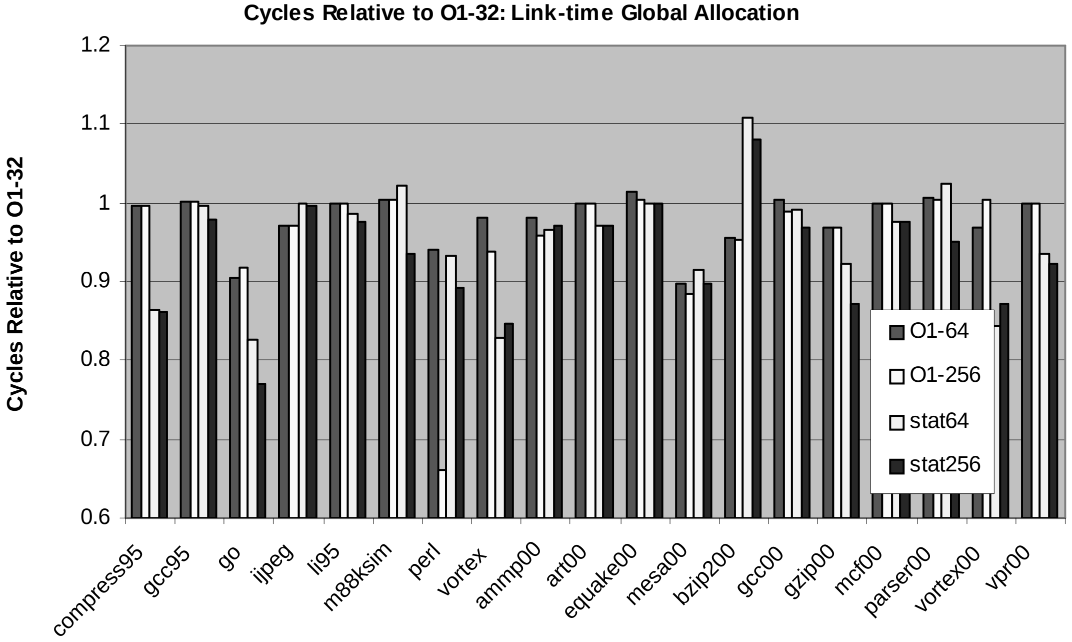


Figure 3.5: Performance of post-link allocation.

As the number of registers is increased for promotion, results generally improve slightly. Some benchmarks, such as `m88ksim` and `vortex`, show a performance degradation. This is due to the extra register save and restore operations at function call boundaries. This overhead outweighs the benefit of promotion. While the reduction in the number of memory operations for `go`, `ijpeg` and `equake` is significant (10%-15% of each category), performance only improves by about 5%-8%.

Link-time Global Variable Register Allocation

Figure 3.5 shows the performance results when link time allocation is applied. The results reported are for 32 and 224 registers specifically set aside for global variables. Because of pre-compiled libraries, we could not permanently allocate any global variables to registers 0-31. This optimization reduces the number of load instructions anywhere from 0% to 50%. Some benchmarks, such as `ijpeg`, `vortex` and `equake` see little or no reduction in memory operations. For those benchmarks that do see a significant reduction, performance improves up to 15%. The corresponding reductions in load and store operations are shown in Figure 3.6 and Figure 3.7.

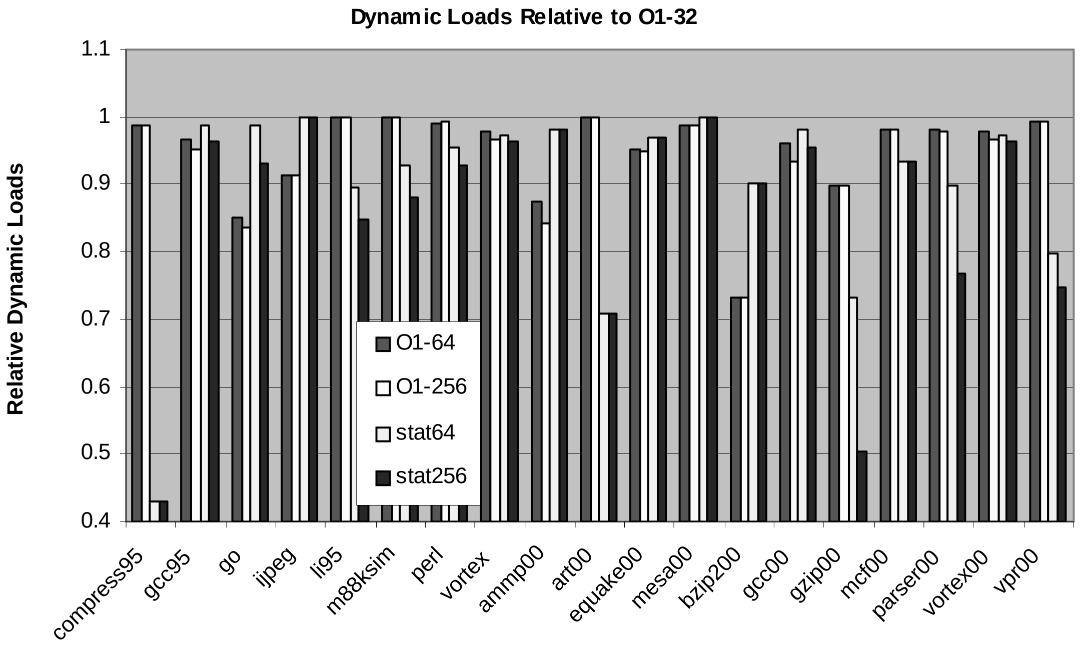


Figure 3.6: Loads eliminated by link-time global variable allocation.

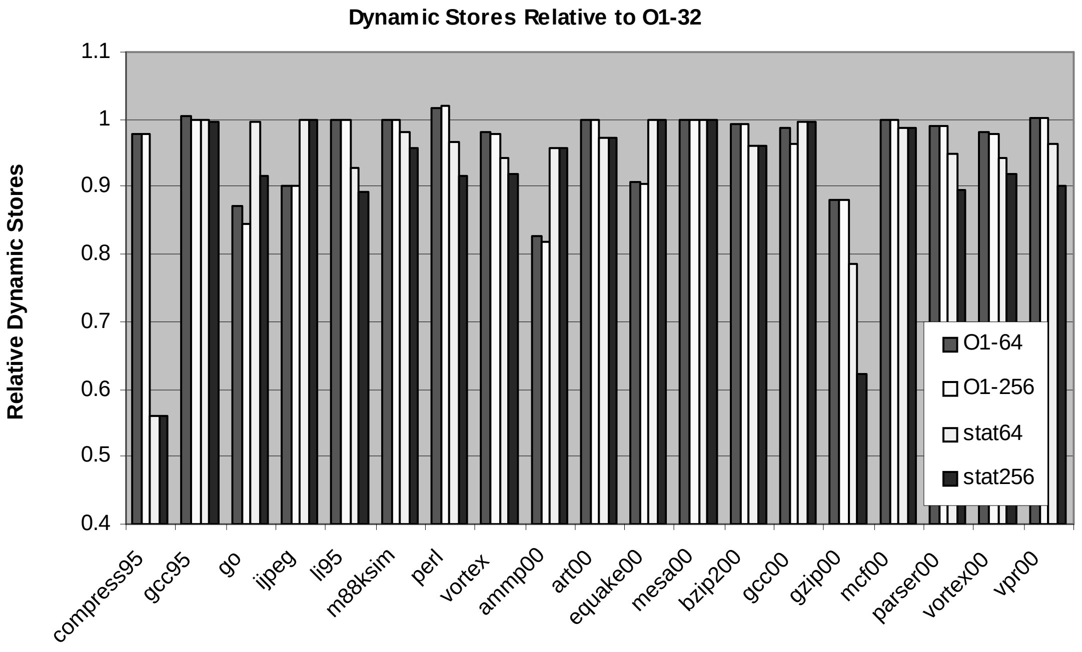


Figure 3.7: Stores eliminated by link-time global variable allocation.

Performance actually degrades in some cases when the number of registers set aside for global variables is increased from 32 to 224 (the stat64 and stat256 configurations, respectively). This is not intuitive and indeed is due to another effect: code layout.

For example, for `vortex` compiled with -O1 optimizations, the instruction cache miss rate is 7%. For the same program compiled with link-time allocation, the instruction cache miss rate is over 9%.

The effect of allocating base addresses is significant for some benchmarks, particularly `go`, which accesses a lot of global arrays. In this case, loads and stores are reduced as well as address arithmetic. Essentially, the base address has been memoized into a register. This reduces the amount of redundancy in computation that is seen during program execution.

`Compress` is not a particularly good benchmark to study in this context because most of the benefit of link-time allocation comes from two global pointer variables which index into I/O buffers. These are used in two small helper functions called `getbyte()` and `putbyte()`, which were introduced when the program was made into a benchmark. Link-time allocation helps to remove this inefficiency and eliminates almost 60% of the load instructions and nearly 50% of the store instructions in that benchmark.

Many of the benchmarks show that adding registers for global variables is much more important than adding registers for local variables. For example, the `compress` benchmark does not benefit at all at the -O1 optimization level with 64 or 256 registers. However, setting aside 16 registers for globals vastly improves the performance. This is true for `go`, `m88ksim`, `art`, `gzip`, and `vpr`, and to a lesser extent some of the other benchmarks.

Adding registers for locals does impact performance by 5% or more for `go`, `m88ksim`, and `equake`. Sometimes variables are aliased for a small initial portion of their overall lifetime and could be allocated permanently to a register afterwards. For example, we noticed that in the `compress` benchmark, the global `seedi` integer could be allocated to a register after it is initialized through a pointer. However, this yielded only a marginal performance improvement, so we did not investigate this further.

Link-time allocation is much more effective than register promotion at improving performance. This points out the importance of permanently allocating globals into registers. Promotion can achieve some of this benefit by removing loads and stores from loop bodies, but it still shuttles data back and forth to memory at region boundaries. If the promotion itself occurs within a function that is called from a loop, the promotion and demo-

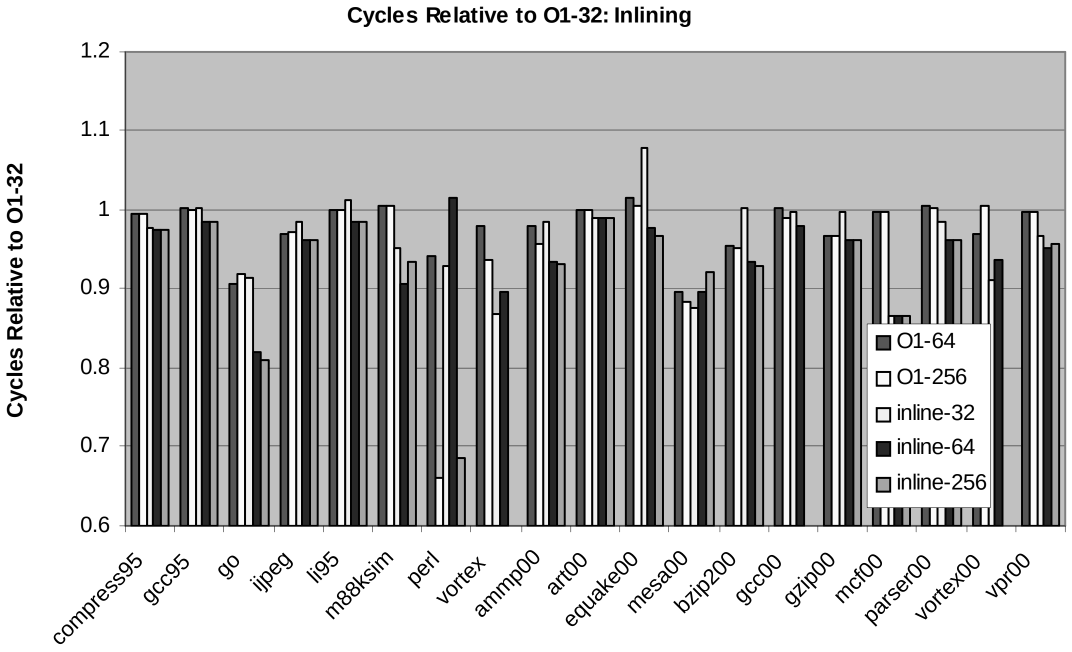


Figure 3.8: Performance of inlining.

tion are still within a loop. Link-time allocation avoids all such unnecessary data movement.

3.5.2. Inlining

Figure 3.8 shows the experiment results when inlining is applied to the benchmarks. The results suggest that with inlining, register pressure increases significantly. This is particularly true for `go` and `equake`, as adding more registers results in performance improvements. Other benchmarks like `compress` and `art` do not make use of extra registers. Figure 3.9 and Figure 3.10 show the reductions in load and store operations due to inlining.

Inlining increases register pressure in two ways. First, the local variables for the caller and callee are combined and allocated in a single graph coloring phase. Both sets of variables compete for the same set of registers. The function call overhead that takes care of shuttling these two sets of variables in and out of registers has been eliminated. Second, inlining also increases the scope of optimization. Because the optimizer has an increased amount of program context, it can perform more transformations. As a general rule, trans-

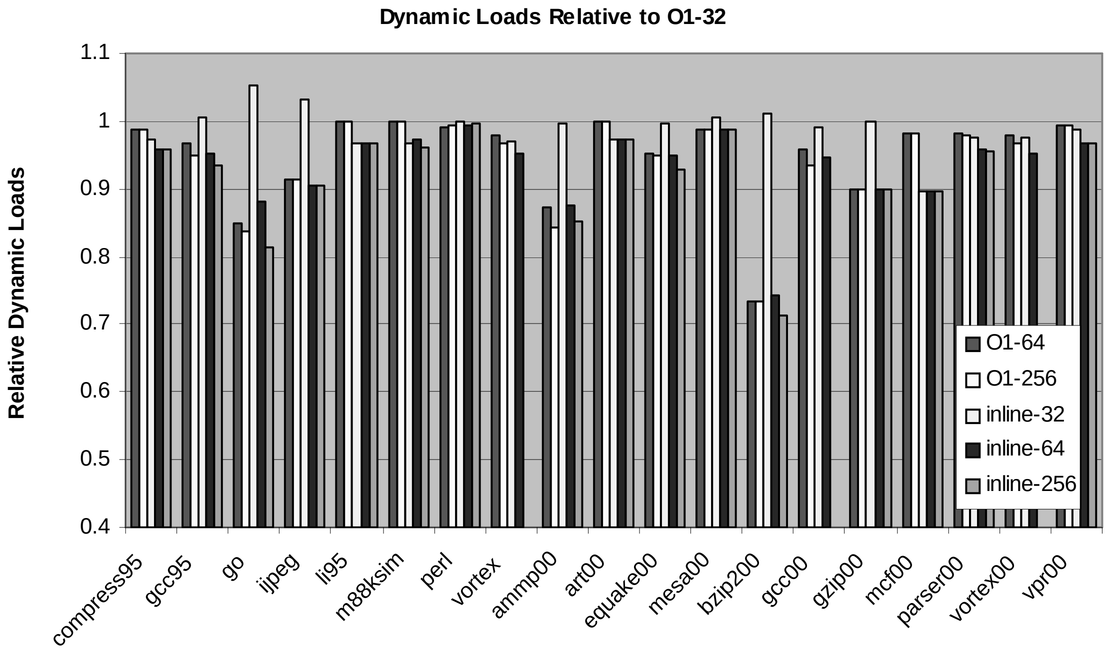


Figure 3.9: Loads eliminated by inlining.

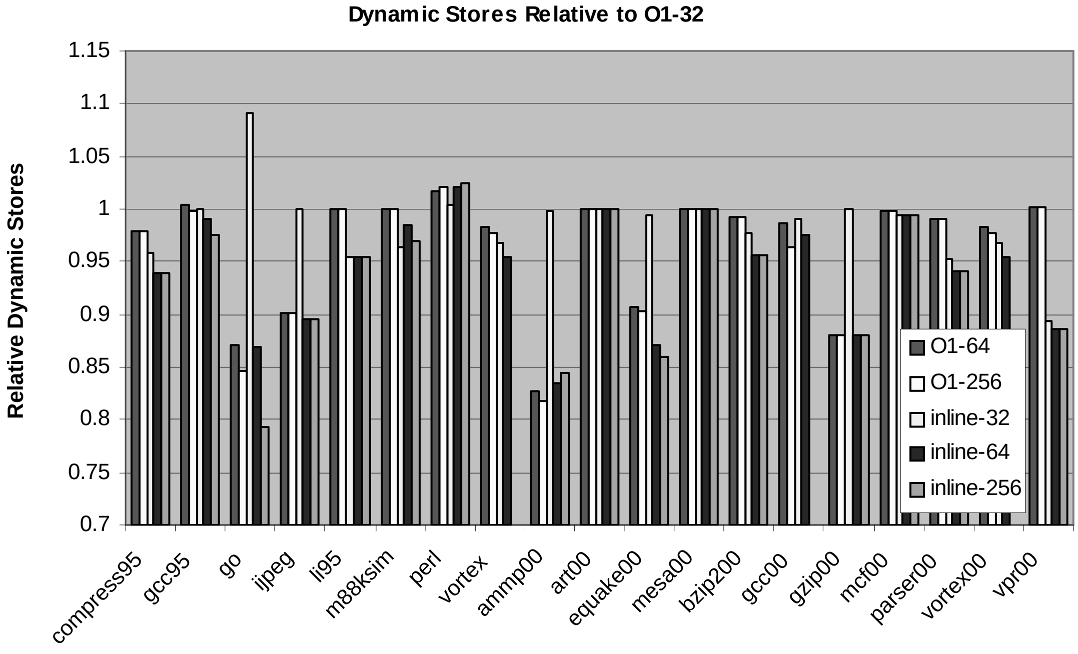


Figure 3.10: Stores eliminated due to inlining.

formations increase register pressure by adding temporaries and extending value lifetimes. As a result, inlining increases register pressure indirectly by opening up more opportunities for program transformation.

Without extra registers, `gzip` perform worse when inlining is performed. This is mainly due to register spilling. Inlining introduces too many simultaneously live variables which must subsequently be spilled. Often this spill code is more expensive than the function call register save and restore because it occurs in multiple places (everywhere the variable is referenced).

3.5.3. Combined Results

The graphs in Figure 3.11 show what happens when we combine the optimizations of the last three sections: register promotion, link-time allocation, and inlining. The third bar shown is what we call “best16”. This includes all the aforementioned optimizations. The ‘16’ in the name indicates that we allowed 16 extra registers for local variables (for promotion and inlining) and 16 extra registers for global link-time allocation, for a total of 32 extra registers. Similarly for best112, where a total of 224 extra registers are allowed, half for local variables and half for link-time allocation. The number of cycles required to execute the program is generally by 10% to 20% in almost every case.

In particular for the `go` benchmark, when best112 is applied, the execution time and number of dynamic instructions has been reduced by 20% over the baseline -O1 optimization level. The number of loads has been reduced by over 30% and the number of stores by over 50%.

Each of the optimizations has increased register usage and takes advantage of extra registers. For example, the extra 32 registers available in the best16 configuration demonstrate significant performance improvements over -O1 optimization. In some cases, like `compress`, `go`, `m88ksim`, and `vpr`, the best112 configuration (a total of 224 extra registers) shows 3% to 7% performance improvement over the best16 configuration. This configuration serves to illustrate the point that more than 32 registers are required to achieve the best possible performance in our compilation environment. Ideally, an architecture should have at least 64 registers to allow for the maximum benefit for register promotion, global variable allocation, and inlining. With more aggressive alias analysis, optimizations, and scheduling, we are confident that the compiler could use even more registers than reported here. Figure 3.12 and Figure 3.13 show the reductions in loads and

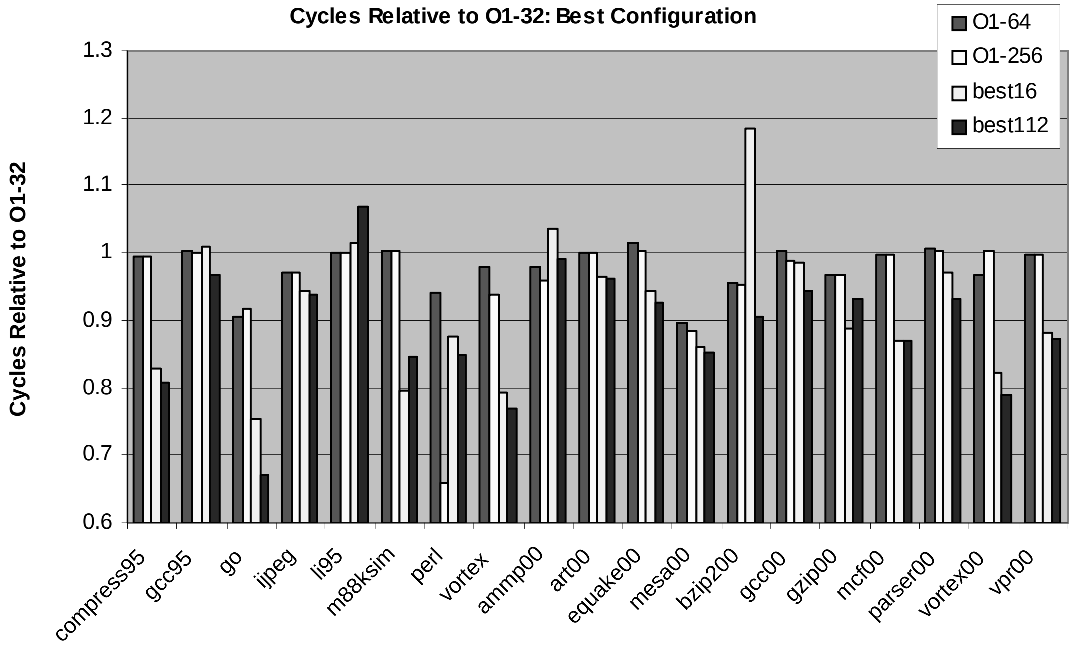


Figure 3.11: Performance of the "best" configurations.

store in this configuration. It is interesting to note the compress benchmark, where over 60% of the load instructions are eliminated by the optimizations.

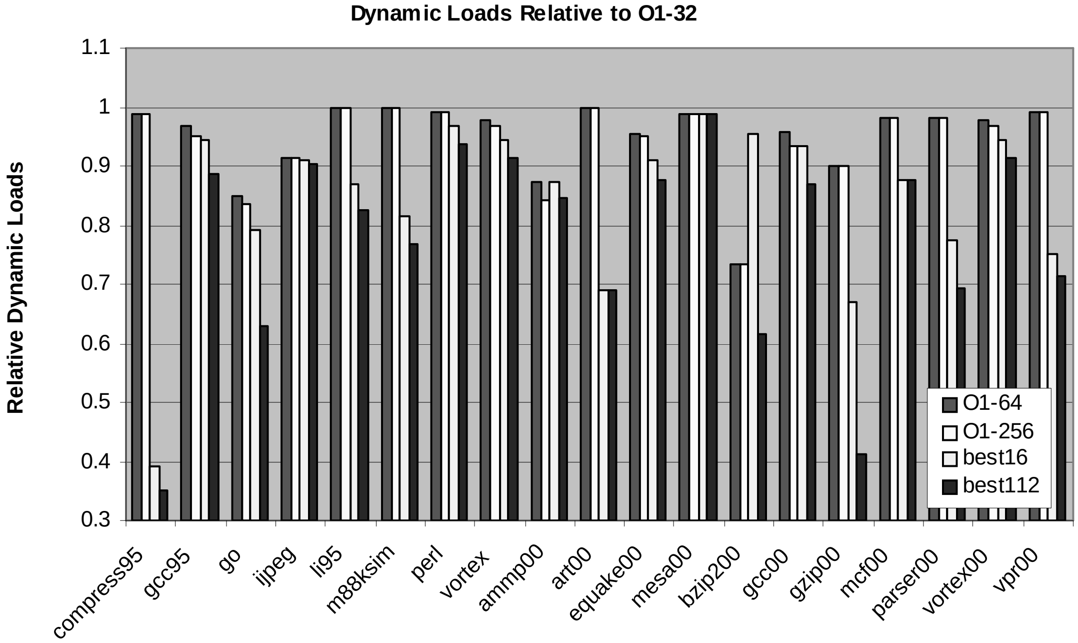


Figure 3.12: Load instructions eliminated by best.

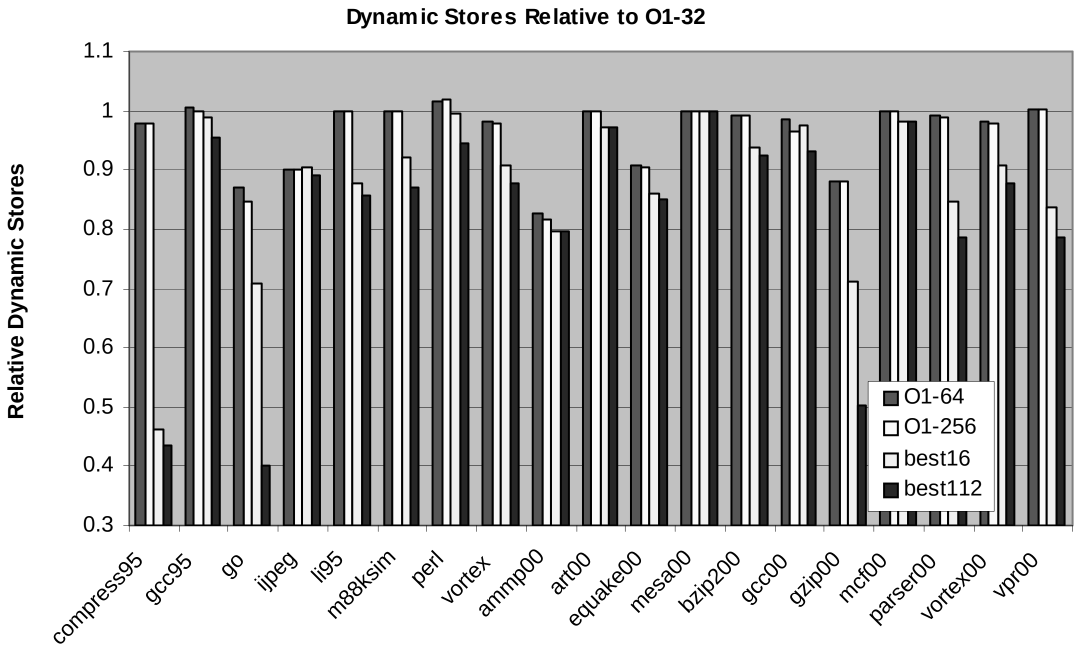


Figure 3.13: Store instructions eliminated by best.

3.5.4. Data Cache Effects

The number of data cache accesses for the best configurations is shown in Figure 3.14. The graph shows the number of data accesses made to the cache. This is an important metric because it shows the effectiveness of register allocation techniques in shielding the memory hierarchy from traffic. The combination of optimizations produce substantial reductions in cache accesses for `go`, `art`, `gzip`, `vpr`, and to a lesser extent the other benchmarks.

There are several potential benefits of reducing the number of cache accesses. For example, fewer demand accesses to the cache by the processor might allow the memory hierarchy to do more effective prefetching since it has more time to query the cache to see if prefetch candidates already reside in the cache. Another important benefit is power savings.

In Table 3.5, we show the reductions in data cache hits and misses for the best112 configurations. The number of cache *misses* is not reduced substantially for any benchmark. On the other hand, the number of cache *hits* has been reduced in most cases, some-

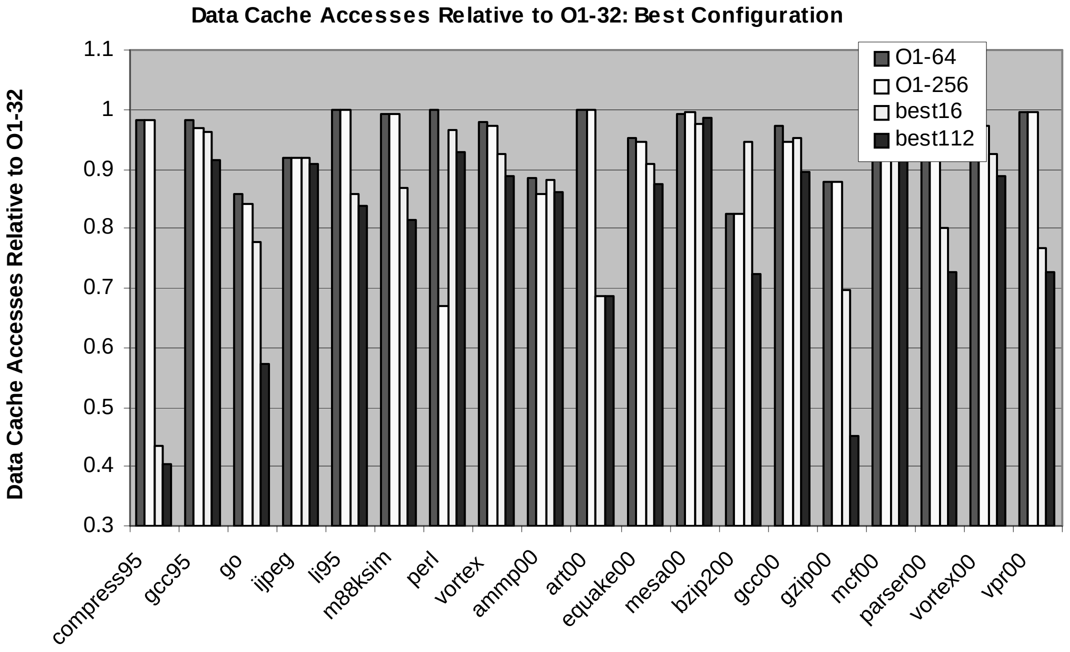


Figure 3.14: Data cache accesses in O1 and best various configurations.

times as much as 30% to 60%. In effect, the compiler has moved items from the data cache into the register file: those items that were already hits in the data cache are now hits in the register file. This is a good optimization because the register file will likely be much faster to access than the cache, in addition to having a much higher bandwidth because of the larger number of ports typically available on a register file.

This is an important illustration of how it is not always most important to optimize the primary “bottleneck” in the system. In the case of memory accesses, it is intuitive to focus on reducing the cache miss latency or miss ratio because misses are expensive in terms of cycles. Such is the focus of work on prefetching and smarter cache replacement policies. Instead, our research optimizes the common case, cache hits, and effectively reduces the cache hit time by eliminating the cache access altogether. Our data shows that techniques which reduce cache miss latency or miss ratio are orthogonal to ours, that is, they can be applied in addition to our optimizations to further improve performance. Processors with larger caches which have two or more cycle access time would benefit even more from optimizing the cache hit time this way. This is discussed in Chapter 6.

For example, if a load-use instruction pair executes in back-to-back cycles on a conventional microarchitecture, changing the load into a register access can improve performance by eliminating the load instruction (one cycle) and also allowing the consumer instruction to start one cycle earlier, for a savings of two cycles. Compiler scheduling may reduce this benefit by separating the load from its use. On the other hand, the situation may be worse if the load has to wait for a free port on the data cache. Allocating the variable to a register instead solves two problems at once: it reduces path length by elimination of load and store instructions, and it solves the bandwidth problem because the register file will typically be highly ported.

Category	Benchmark	Misses in best112 Relative to Baseline	Hits in best112 Relative to Baseline
SPECint95	compress	104%	38%
	gcc	117%	91%
	go	100%	57%
	ijpeg	99%	91%
	li	101%	83%
	m88ksim	100%	81%
	perl	100%	93%
	vortex	105%	89%
SPECfp2000	ammp	100%	85%
	art	100%	56%
	equake	98%	87%
	mesa	99%	99%
SPECint2000	bzip	100%	72%
	gcc	121%	89%
	gzip	101%	43%
	mcf	100%	92%
	parser	100%	72%
	vortex	105%	89%
	vpr	98%	72%

Table 3.5: Data cache misses of the best configurations relative to O1-32

3.5.5. Instruction Bandwidth and Cache Effects

The optimizations we have shown in this chapter have a significant impact on the performance of the frontend of the processor. In this section we discuss some of these.

Figure 3.15 shows the dynamic instruction counts after optimization, again relative to the O1-32 baseline. The reductions range from nothing up to 30%. Most of the benchmarks reduce instruction count by a modest 5 to 10%. This is an indicator of the number of

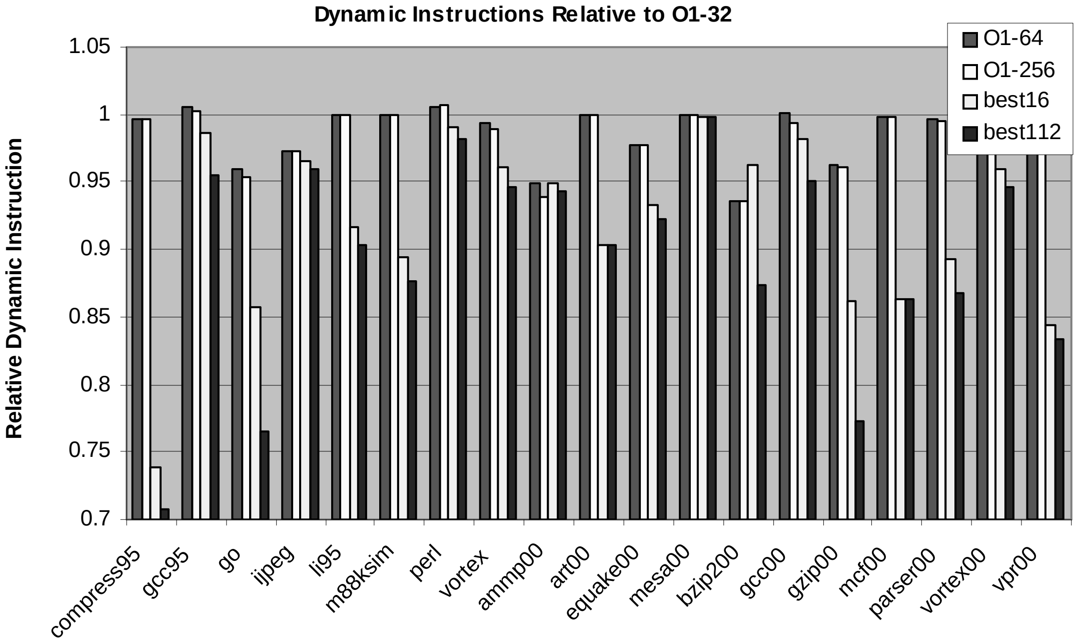


Figure 3.15: Dynamic instruction reduction due to best optimizations.

instructions removed from the important sections of the binary. As a result, instruction cache pressure is less, making the cache effectively bigger since it can hold more of the program.

In Figure 3.16 we show how inlining and our “best” optimization impacts instruction cache performance. Instruction cache performance is not significantly impacted by our optimizations in most cases. Inlining often results in a better cache hit rate than -O1 optimization. The exceptions are `li95`, `m88ksim`, and `vortex`, where the instruction cache miss rate increases by 1% or more. When the “best” optimizations are turned on, the situation is remedied because the best configuration does not do inlining as aggressively since some registers are reserved for link-time global allocation and thus are not available for inlining. Alternative code layouts were not considered, although our experience indicates this could improve performance by several percent.

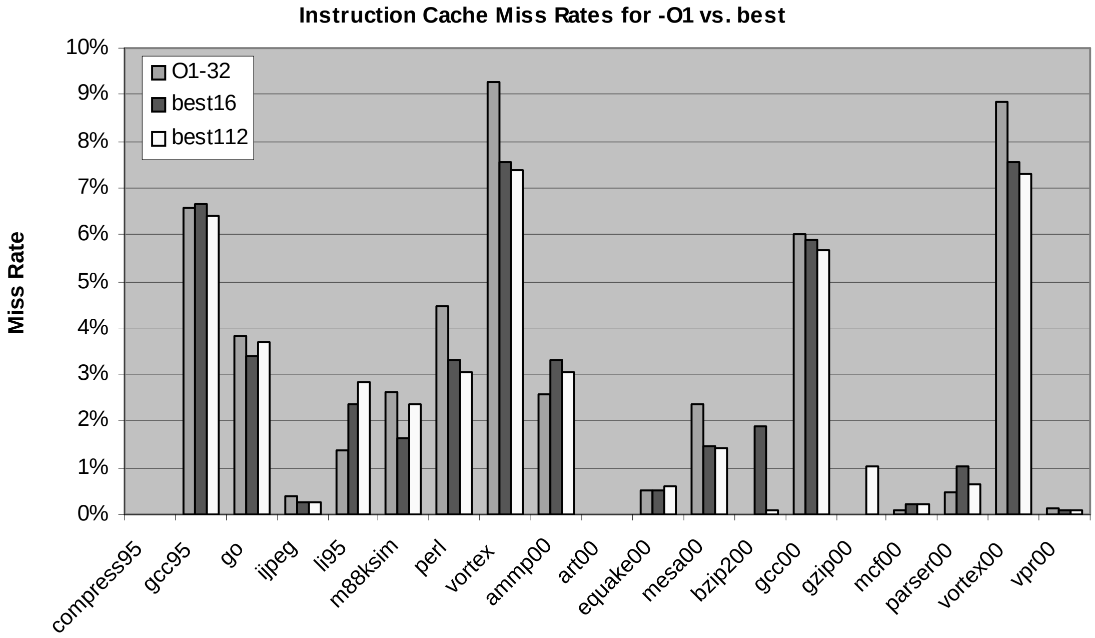


Figure 3.16: Instruction cache miss rates for the -O1-32 and best configurations.

3.6. Theoretical Register Requirements

This subsection examines several measurements to determine what the most appropriate number of registers would be for an architecture that processes general purpose code.

3.6.1. Variable Count Arguments

The first measurement we show is a simple counting argument pointing out that there are many candidates for register allocation. Additional intraprocedural optimizations, such as our -O2 configuration, generally increase the number of candidates for allocation. Inlining in the -O3 column shows a significant reduction in the number of candidates. This is because inlining eliminates parameter variables and allows propagation optimizations. At the same time, inlining increases the number of long-lived variables that are allocated to registers.

Category	Benchmark	-O1 Local Variables	-O2 Local Variables	-O3 Local Variables	-O4 Local Variables
SPECint95	compress	117	120	49	49
	gcc	24842	25102	21318	21318
	go	3435	3490	1948	1948
	ijpeg	3321	3330	2251	2251
	li	1703	1713	1506	1506
	m88ksim	1859	1887	1298	1298
	perl	4062	4125	3764	3764
	vortex	13690	13778	7859	7860
SPECfp2000	ammp	2332	2364	1855	1855
	art	154	254	174	174
	equake	191	218	84	84
SPECint2000	gcc	27289	27579	23048	23048
	gzip	690	714	398	398
	mcf	218	218	76	76
	parser	1924	2001	1296	1296
	vortex	13690	13777	7281	7860
	vpr	2244	2388	1639	1639

Table 3.6: Statistics on local variables under different optimizations levels.

Table 3.7 shows the number and types of global variables in the benchmarks. The fourth column in the table shows the percentage of global variables that are not considered as candidates to be placed into a register. This percentage is computed by linking together the benchmark at the MIRV IR level and processing the MIRV IR with a filter that determines whether the global variable's address is ever taken. If it is, then we say the variable cannot go into a register (even though it may be able to be enregistered for parts of its lifetime).

The integer benchmarks fall into two categories. For the majority of the benchmarks, most global variables could be allocated for their entire lifetime. Two interesting cases are `go` and `vortex`, where there is a high percentage of globals that are aliased. In both benchmarks, most of the variables in question are passed by address to some function which changes the global's value. The number of call sites where this happens is usually fairly small for any given variable. In vortex, a heavily used variable called Theory is modified through a pointer in memory management code. It appears that this use through the pointer is for initialization only (so register promotion could promote the global to a register after initialization is over). The floating point benchmark equake shows similar behavior.

Overall, there are a significant number of global variables in these benchmarks which may benefit from register allocation. The global aggregates (shown in the fifth column of Table 3.7) are the number of arrays and structures found at the global scope. The base addresses of these structures are candidates for register allocation as well.

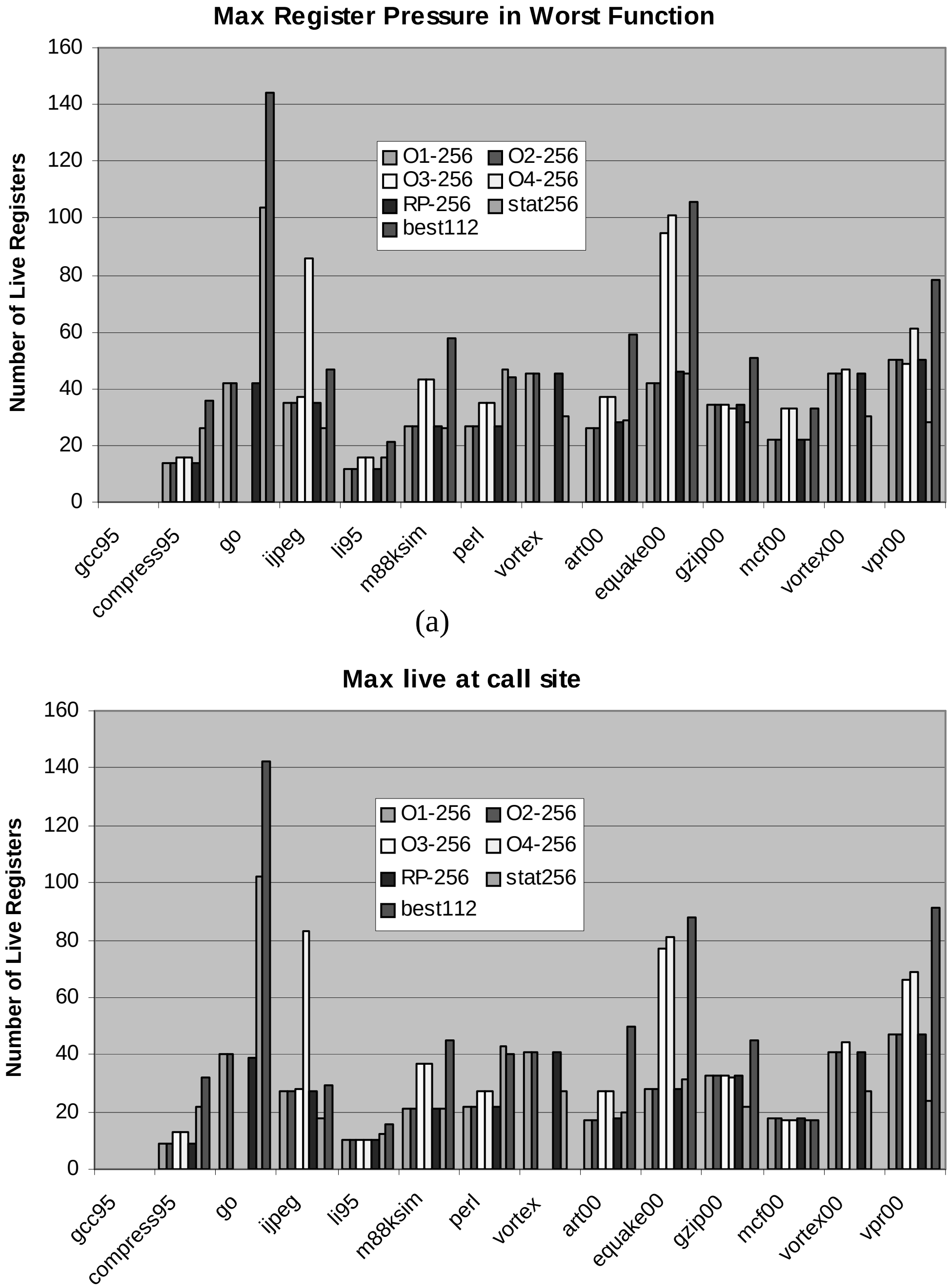
Category	Benchmark	Global Variables	Un-allocatable Globals	Global Aggregates
SPECint95	compress	27	3.7%	14
	gcc	1018	7.1%	298
	go	80	15%	200
	ijpeg	29	3.4%	35
	li	79	0.0%	3
	m88ksim	106	1.9%	8.7
	perl	208	2.9%	42
	vortex	515	41.7%	105
SPECfp2000	ammp	48	6.3%	4
	art	28	7.1%	4
	equake	34	35.3%	4
SPECint2000	gcc	1093	7.1%	340
	gzip	99	2.0%	47
	mcf	6	0.0%	3
	parser	83	22.9%	92
	vortex	515	41.7%	105
	vpr	96	6.3%	10

Table 3.7: Statistics on global variables under -O1 optimization.

Figure 3.17 shows two more counting arguments. Figure 3.17(a) shows an estimate of the maximum register pressure in the “worst” function in each of the benchmarks. Typically under 40 registers are required by such functions, but occasionally the number climbs well above 50, particularly under the -O4 optimization level. A similar situation is seen in Figure 3.17(b), where the maximum live at any function call site is shown. For example, there are 40 simultaneously live values at the “worst” call site in the go benchmark under -O1 optimization.

3.6.2. Instruction Count Argument

The last section showed measurements of the “worst” functions in the benchmarks but said nothing about the average register requirements. This section shows data on the



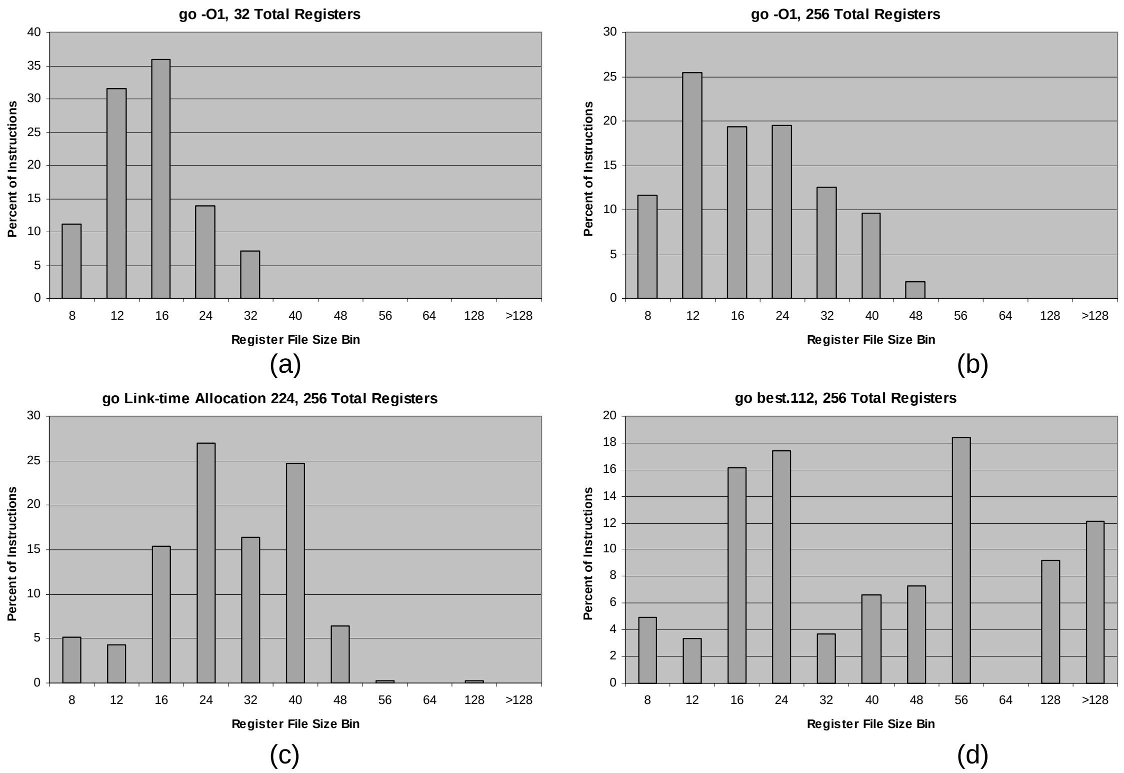
(a)

(b)

Figure 3.17: Other estimates of register pressure in the benchmarks.
Note the change in y-axis scale.

register requirements of the go benchmark over four different optimization levels in Figure 3.18.[2] The X-axis of each graph is the number of registers in a register bin. The Y-axis shows the percentage of dynamic instructions that execute in functions which required

[2] We have not shown these graphs for every benchmark due to space considerations.



(a)

(b)

(c)

(d)

Figure 3.18: Register requirements for go across several optimization levels.
The X axis is a register file size bin, where 8 represents 0-8 registers, 12 represents 9-12, 16 represents 13-16, etc.

that number of registers. For example, in Figure 3.18(a), 36% of the instructions come from functions where 13-16 registers are desired by the function. This does not count special registers such as the frame pointer, stack pointer, global pointer, zero, and assembler and kernel reserved registers. We will assume that there are 8 such reserved registers for the sake of argument. For these instructions, a register file size of 13+8=21 to 16+8=24 is the right number.

These numbers are produced by annotating the each function with a *maxlive* attribute which tells how many registers are simultaneously live in the function. This *maxlive* number comes directly from the live variable analysis phase of the backend's register allocator. Its final value is computed immediately before the assembly code is generated, so it includes the effect of spilling code to reduce register pressure for a given register configuration. This information is used along with a profile that tells how many instructions were executed from each function. Note that the *maxlive* number is not how many colors were required to color the graph, but instead is the maximum number of values that are simultaneously live. It would take at least this many registers to color the graph without any additional spilling code. Therefore, *maxlive* is a lower bound on the number of registers required for maximum performance.

The executable that was used to produce Figure 3.18(a) was compiled assuming 32 registers were available on the machine. Figure 3.18(b) shows that when this assumption is changed, to 256 registers, the profile looks significantly different. In this case, 32 registers are required over 10% of the time and 40 registers another 10%. This points out that when the compiler is limited to 32 registers, the register-usage profile appears to be very conservative (because of the spilling code inserted during register allocation). Only when the compiler is loosed from the restriction of a small register file can a true measurement of register requirements be made.

Thus far, we have shown that with traditional optimizations such as CSE and LICM, the MIRV compiler could easily and profitably use a machine with 40 or more registers.

Figure 3.18(c) shows what happens when link-time allocation is applied to go. Even more registers are required–25% of the time, a register file size of 32-40 is appropriate. Figure 3.18(d) shows what happens when all of the optimizations in this chapter are turned on. A large shift to the right of the graph occurs, and many instructions are executed in a context that requires 56 or more registers. From this data it is clear that current compiler technology can utilize a large number of registers. At least 64 registers are required for the benchmarks shown here.

3.6.3. Cross-Function Argument

The previous sections considered register allocation in the conventional sense, where the allocator is limited to candidates within a single function. A running program, however, typically has several functions “in progress” on the call stack. Each of these has some number of live values which should be summed to find out the best number of registers for the architecture. Since this is a hard number to compute, in this section we provide a couple of estimates to show the utility of a cross-function allocation scheme.

The first is simply the call graph depth, shown in Figure 3.20. This graph shows the maximum call depth reached by the benchmark. Most benchmarks only traverse 10 to

20 levels deep in the call graph. This supports the classic argument for register windows, which is that windows will allow the machine to capture most of the dynamic call stack without stack overflows and underflows, which necessitate memory operations to manage the stack.

The graph in Figure 3.20(b) shows our estimate of the register requirement of each of the benchmarks. This estimate is produced by applying the following formula to the dynamic call graph:

$$CGmaxLive(\text{caller}) = MAX\begin{pmatrix} CGmaxLive(\text{callee}) + maxLiveAtCallSite(\text{caller}), \\ maxLive(\text{caller}) \end{pmatrix} \quad \textit{(Eq. 2)}$$

The formula is applied during a run of the benchmark where the call graph is annotated with two pieces of information. At each call site, we have placed an integer for the number of live variables at the call site minus the number of parameters. We call this number *maxLiveAtCallSite*(caller) and do not include parameters under the assumption that *CGmaxLive*(callee) will include those. The other number is annotated into the function itself and is the most number of live variables at any point of the caller function. We call this *maxLive*(caller).

As an example of how this formula works, consider the following situation. Function A has *maxLive*(caller) = 11 and *maxLiveAtCallSite*(caller) = 3. If we call a function B whose *CGmaxLive* is 6, then the simulator says that for this sequence of function calls, max(3+6, 11) = 11 registers is plenty. If B's *CGmaxLive* is 9, then the 12 registers is about right.

The graph in Figure 3.20 shows this estimate across our benchmark. A very large number of registers is suggested. When all optimizations are turned on, the "best" number of registers is estimated to be about 100 or more for most benchmarks. Some require several hundred registers. The outlying point is the `li` benchmark. This is a recursive descent program has a very complicated dynamic call graph for which it would be difficult to keep all values in registers.

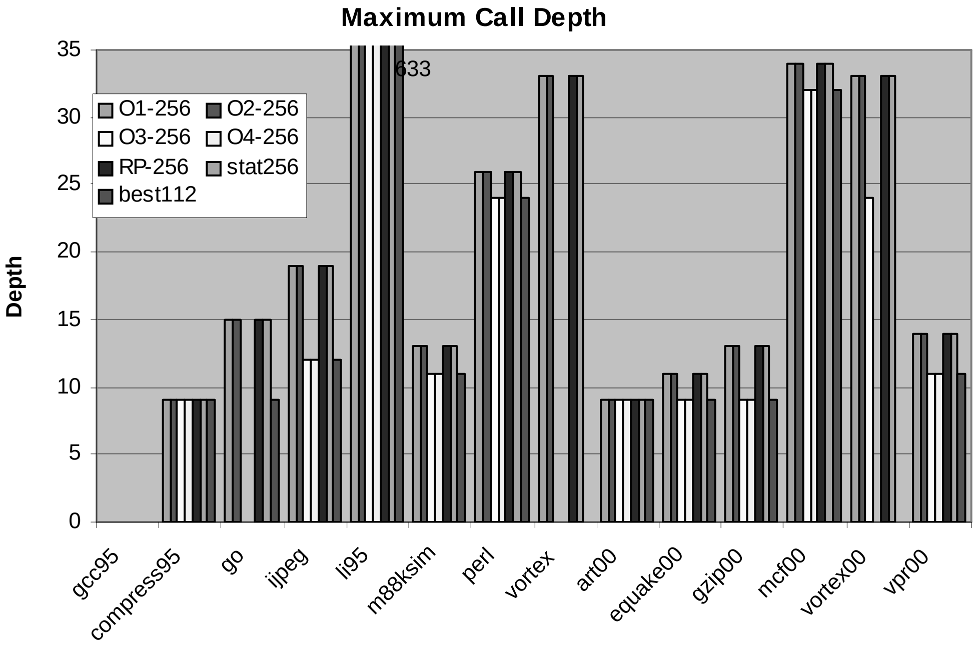


Figure 3.19: Maximum subroutine linkage depth reached in the benchmarks.

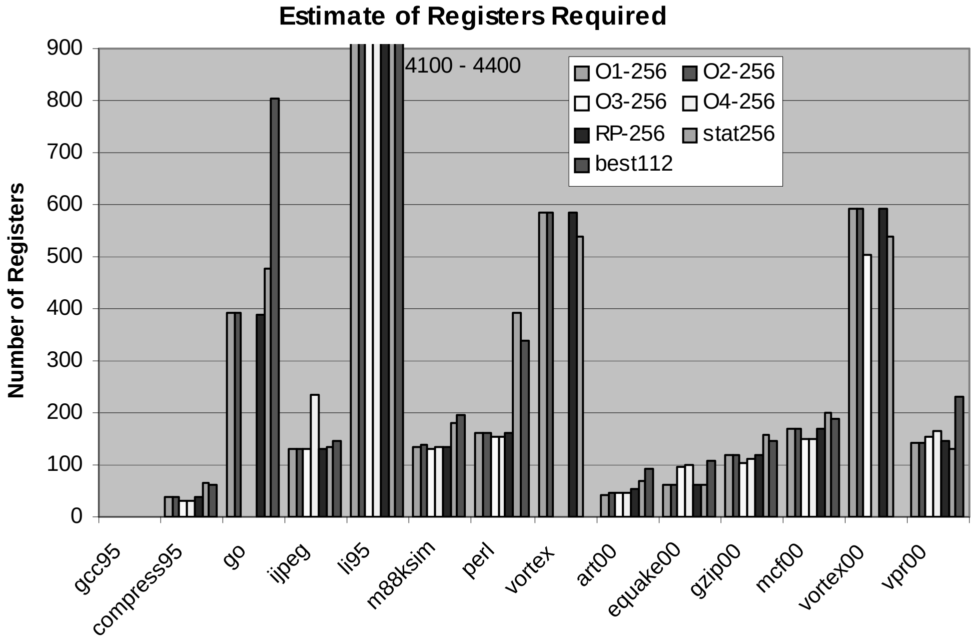


Figure 3.20: An estimate of register pressure based on the dynamic call graph.

3.7. Register Windows Simulations

We modified the MIRV compiler and the SimpleScalar `sim-outorder` simulator to model the register window configuration suggested in the previous section. The compiler uses all registers as if they were callee-saved, eliminates all callee save operations in the function prologues and epilogues, and remove restore operations of the frame pointer, stack pointer, and return address, since the hardware will restore those. The simulator saves the register file contents at every function call and restores them when the function returns. All registers are overlapped at the function call for speed of simulation[3]. The only caveat with our results is that the simulation methodology assumes a well-structured call graph–the perl benchmark, for example, does not work on the simulator because the call graph is not balanced[4]. No modeling is performed for window spilling so the results are optimistic, but they do show an upper bound on the potential register requirements of the benchmarks.

The results are shown in Figure 3.21. There is an average 8% performance improvement, with performance improvements ranging from -6% to 22%. This is significant because it represents overhead instructions inserted by the compiler for the purpose of managing subroutine linkage. Instruction count improves by similar percentages while the number of memory references is more dramatically improved–up to 47%, with an average of 21% reduction. These results are shown in Figure 3.21, Figure 3.22, and Figure 3.23. It is interesting to note that the biggest benchmarks, `gcc` and `vortex` being examples, are most positively impacted by the windowing. We had expected that the overall performance improvements would be more weighty than the memory reference count improvements, thinking that memory references are expensive operations. However, all the memory operations removed by the windowing optimization are to the stack, which is

[3] Typically register windowing schemes overlap a subset of the registers for the purposes of passing parameters to functions. We overlap all of them, which in itself presents an interesting configuration where many registers could be used for parameter passing. This full overlap may also be advantageous in the implementation to save adding the current window pointer to every register specifier; we leave a more in-depth examination for future work.

[4] This can be fixed, obviously, since such benchmarks work on real register-window machines.

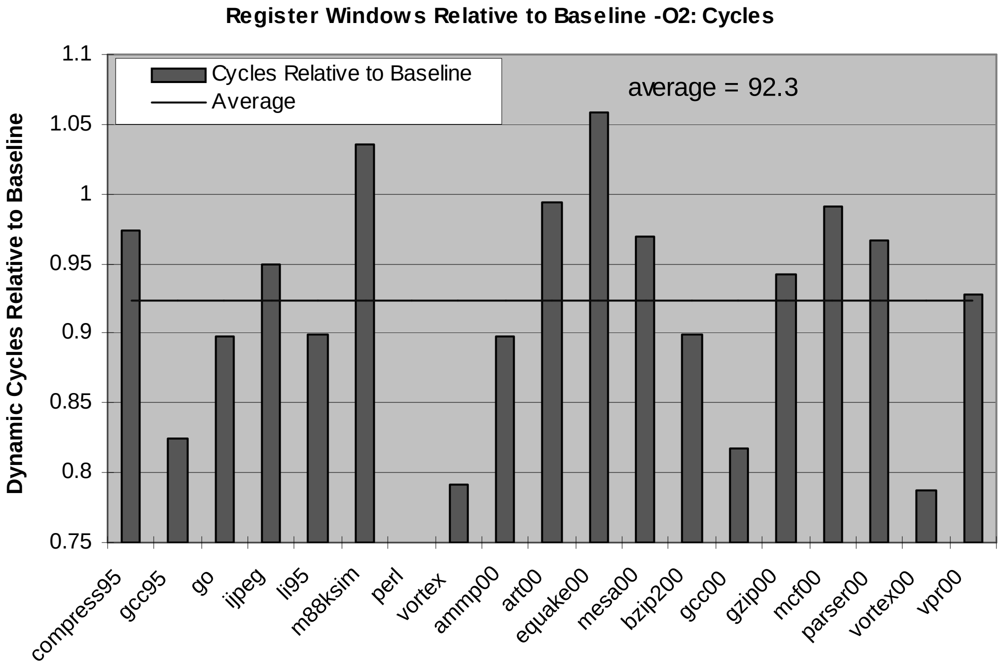


Figure 3.21: Register windows performance compared to -O2.

usually in the cache. Once again, our optimizations have succeeded at eliminating cache hits in favor of the faster register file.

The performance degradations are primarily due to code layout effects. We found that for `m88ksim`, for example, increasing the instruction cache to 64K got rid of the negative performance effect of register windows. Still, code layout can be a significant problem for the instruction cache–even if instructions are removed, the optimized binary could suffer the misfortune of having more cache conflicts than the original.

Whether it is worthwhile to implement this technique, however, is not very clear from this data alone. Figure 3.24 shows additional data that verifies the estimations made in the previous section. Data from the `gcc95` benchmark is shown. The number of register windows concurrently alive are shown, weighted by execution cycles. The graph shows that just over 20% of the execution cycles were spent in a function 8-levels deep on the call graph. Another 20% were spent in functions 9 levels deep. Overall, 80% of the execution time is spent within levels 6 to 12. Therefore, 7 register windows would be sufficient to capture the majority of the execution time in this benchmark. That equates to 7 * 32 = 224 registers since we use full overlap. Of course, not all the functions are going to

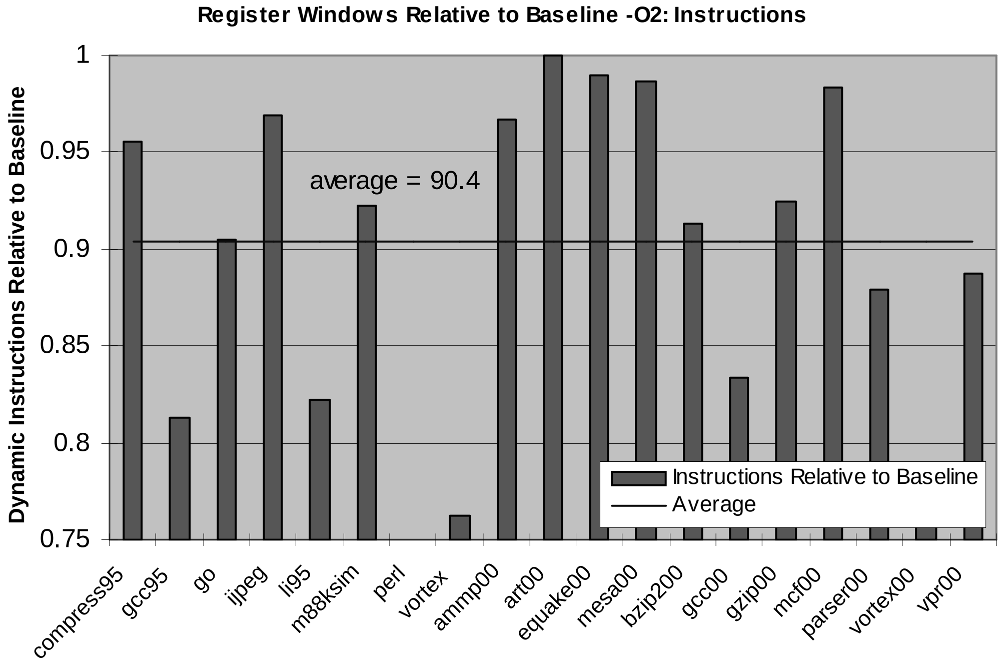


Figure 3.22: Register window instruction count improvement over -O2.

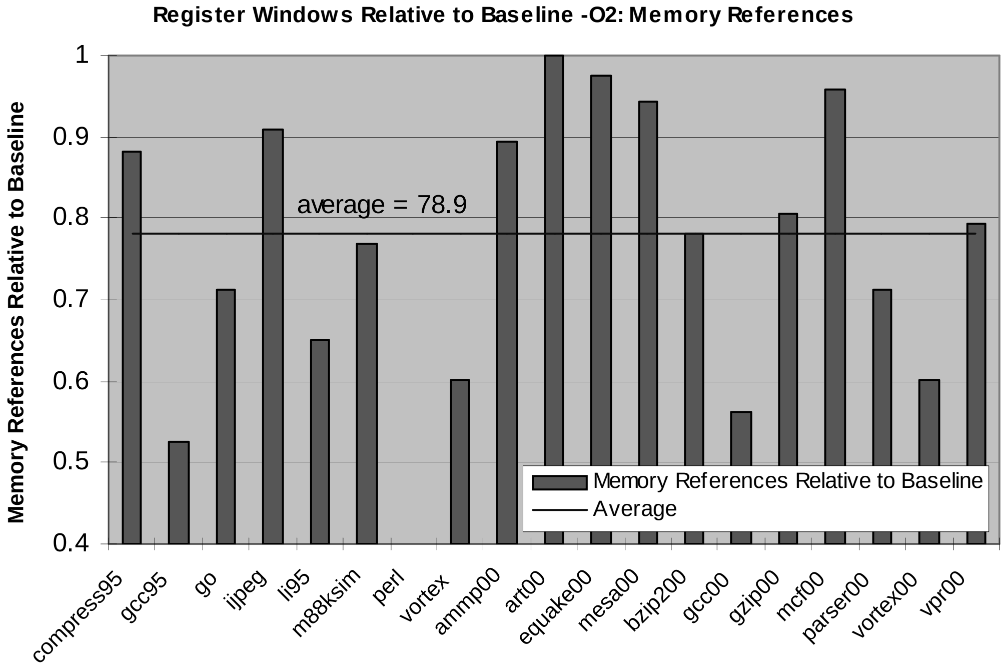


Figure 3.23: Register window memory instruction count improvement.

need a full 32 registers, but different functions may exist at the same level in the call graph. These will have different register requirements.

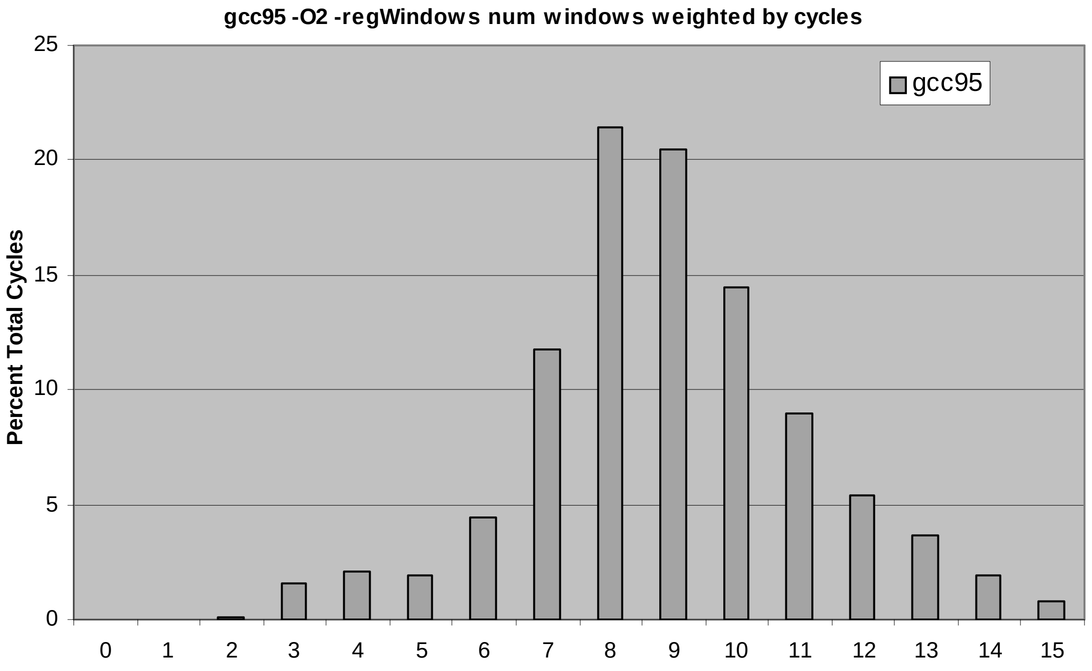


Figure 3.24: The number of register windows concurrently active.

The high-water mark of register windows concurrently used by the benchmarks is shown in Figure 3.25. Most benchmarks require around 30 or fewer concurrently active register windows, at their highest requirements. The `gcc`, `li`, and `parser` benchmarks are exceptional cases.

We also simulated the window model in addition to all of the other optimizations used in this chapter. The performance of this configuration, once again relative to an O1-32 baseline to allowed it to be compared to Figure 3.11, is shown in Figure 3.26. Performance improvements range from almost 15% to over 45%. The windowing improvements are roughly additive with the other optimizations. This is intuitive since they target different kinds of memory operations. The best112 jobs target global scalars through global variable register allocation and conventional register promotion, where most of the benefit is found. Inlining addresses some of the same overheads that register windows do, but inlining is limited in application because of problems with code bloat. The instruction count improves 6% to 30% while the memory references are reduced 11% to 63%, as shown in Figure 3.27 and Figure 3.28.

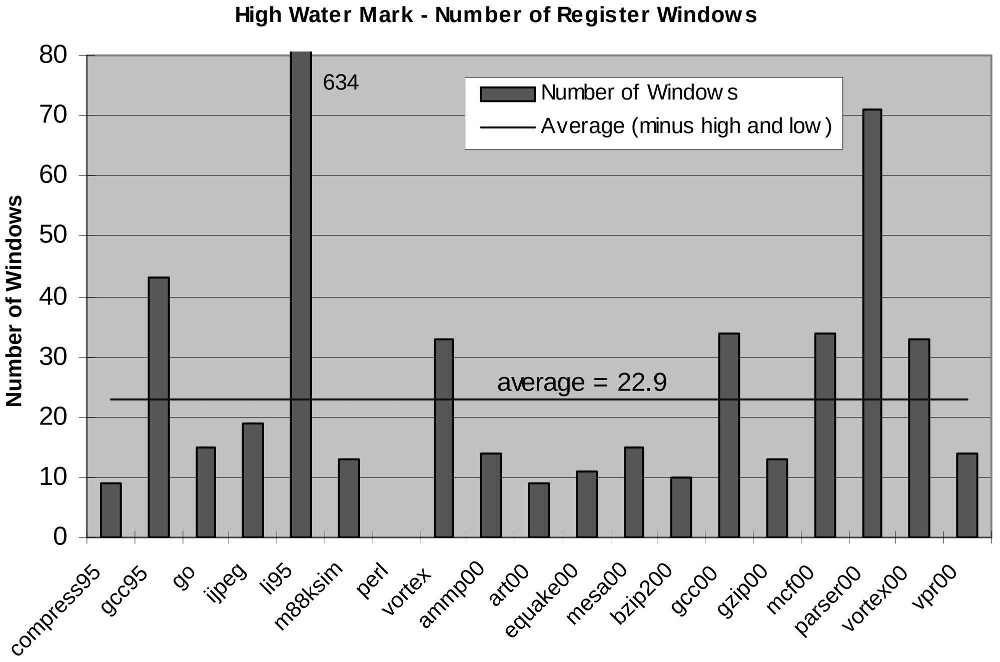


Figure 3.25: High-water mark of number of register windows concurrently active.

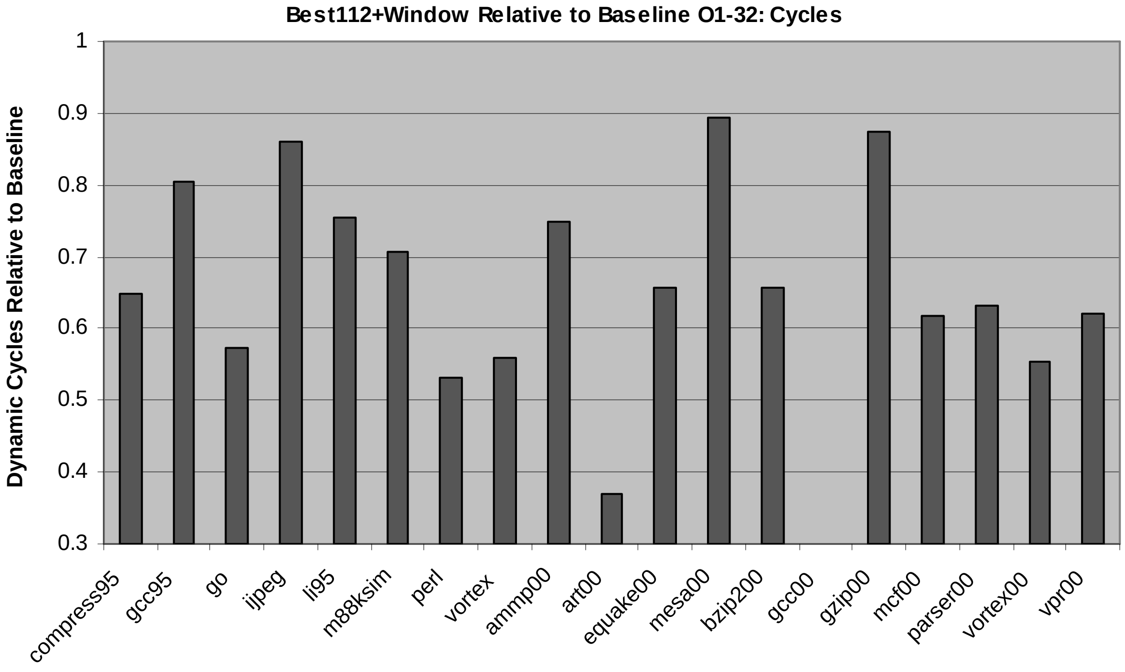


Figure 3.26: Performance of best112+register windows relative to -O1.

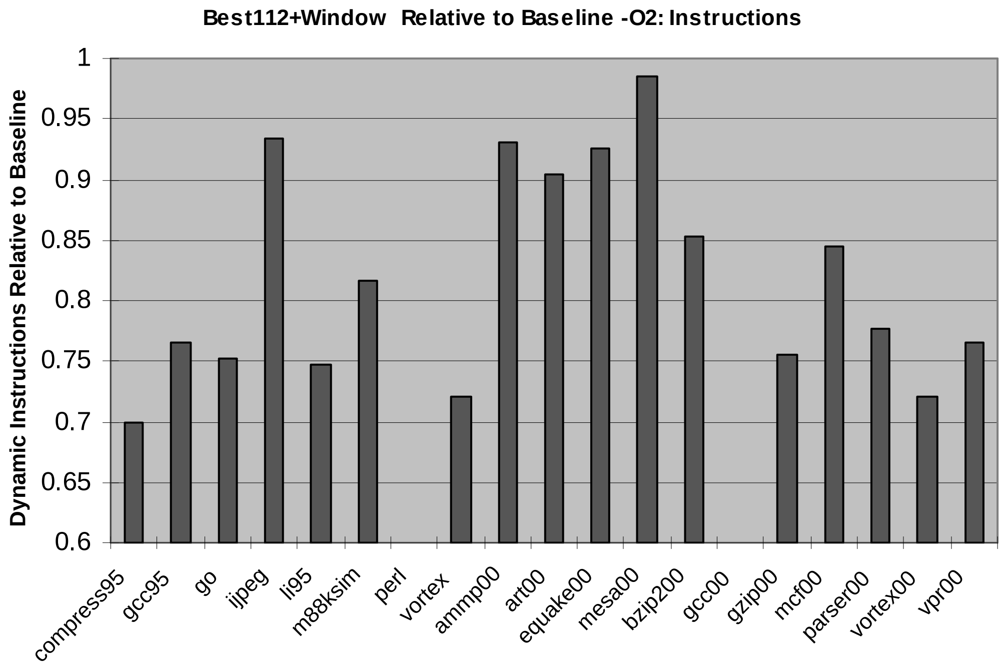


Figure 3.27: Instruction count improvements of best112+register windows.

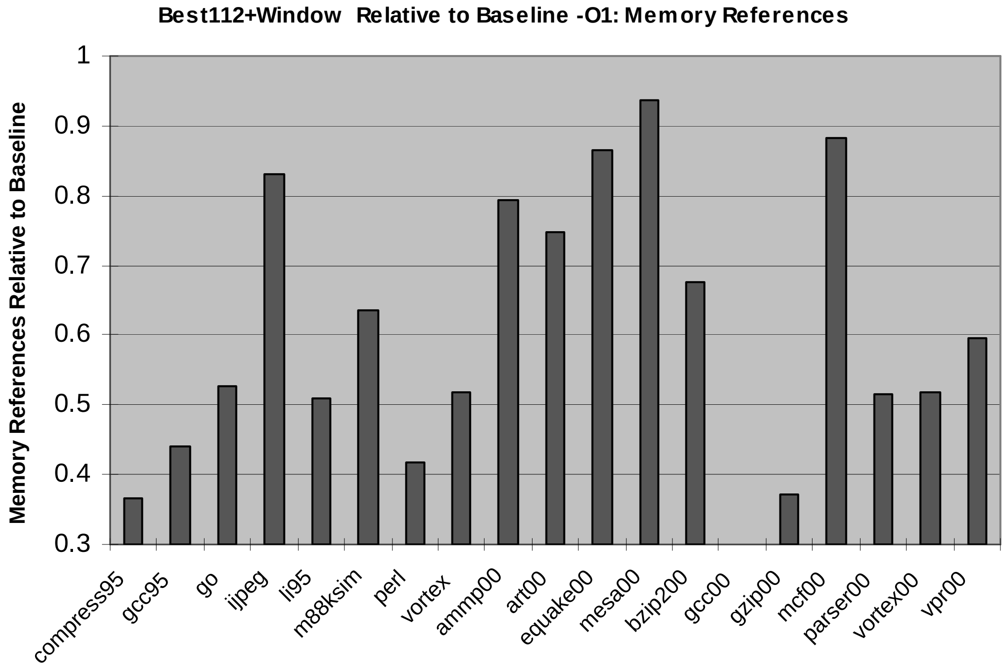


Figure 3.28: Memory reference reductions of best112+register windows.

3.8. Related Work

3.8.1. Intra-Procedural Allocation

Mahlke et. al. examined the trade-off between architected register file size and multiple instruction issue per cycle [Mahl92a]. They found that aggressive optimizations such as loop unrolling and induction variable expansion are effective for machines with large, moderate, and even small register files, but that for small register files, the benefits are limited because of the excessive spill code introduced. Additional instruction issue slots can ameliorate this by effectively hiding some of the spill code. This work noticed little speedup or reduction in memory traffic for register files larger than about 24 allocatable registers (often fewer registers were required). The compiler used in the study was not able to take advantage of more registers because of the conventional application binary interface [SysV91] and lack of optimizations in the compiler such as those described here.

Benitez and Davidson study the effect of register deprivation on compiler optimizations [Beni93]. The technique proposed in that paper is to study the effect of a series of probes where the compiler is deprived of more registers in each successive probe. This allows the compiler writer to examine the effect of optimizations on register utilization. The paper suggests that current technology cannot utilize a large number of registers.

The large body of research into optimizing spill code indicates the prevalence of spill operations in modern programs and highlights the importance of having a sufficient number of registers. For example, Cooper and Harvey propose the compiler-controlled memory which combines hardware and software modifications to attempt to reduce the cost of spill code [Coop98a]. The hardware mechanism proposed is a small compiler-controlled memory (CCM) that is used as a secondary register file for spill code. The compiler allocates spill locations in the CCM either by a post-pass allocator that runs after a standard graph-coloring allocator, or by an integrated allocator that runs with the spill code insertion part of the Chaitin-Briggs register allocator. A number of routines in SPEC95, SPEC89, and various numerical algorithms were found to require significant spill code, but rarely were more than 250 additional storage locations required to house the spilled variables. Potential performance improvements were on the order of 10-15% on a processor with a 2-cycle memory access time, but effects from larger traditional caches, write buffers, victim caches, or prefetching were not modeled. These results show the potential benefit of providing a large number of architected registers–not only simplifying the compilation process, but also reducing spill code and memory traffic.

3.8.2. Inter-Procedural Allocation

Wall described a link-time approach that allocates local and global variables as well as labels into registers [Wall86]. The linker can optionally do a global register allocation pass which builds the call graph of the program and groups functions based on whether they can be simultaneously active or not. For functions that can never be simultaneously active, their local variables can be allocated to the same registers without needing to save and restore them across call boundaries. Functions that can be simultaneously active have their locals allocated to separate regions of the register file, while functions that cannot be live at the same time can share registers. Global variables as well as constants such as array base addresses take part in the link-time allocation as well. Once a global is allocated to a register, it resides in that register for its entire lifetime. The most frequently used variables are selected to be allocated into registers. This is done either with static estimates or with profiling data. Once it is determined which variables will be allocated to registers, the linker rewrites the object code under the direction of compiler-provided annotations. Compared to the baseline—the original code produced by the compiler which has only the 8 temporary registers—the link-time technique improves executable speed by 10 to 25% on a configuration with 52 registers. Link-time allocation (without dataflow analysis or coloring at all) does better than traditional graph coloring such as is done in a Briggs-Chaitin allocator [Chai81, Chai82, Brig92]. Graph coloring can be added in Wall's system in order to allow variables which do not conflict within a procedure to share a register. This local coloring is especially important when there are few registers in the machine but does not improve performance much with 52 processor registers.

Our post-link allocation strategy is closely modeled after an earlier one described by Wall. First, we compare our results against a more aggressive set of baseline compiler optimizations than the previous work. Second, we only consider global variables because we do not have the ability to modify any library code as in the previous study; local variables are allocated with traditional graph coloring techniques within the constraints of the application binary interface. Third, we examine the benefits of global variable register allocation over a wider range of machine configurations; in particular, the number of registers is a variable in this study but was fixed at 52 in the previous work. Fourth, we choose a simpler frequency-based allocation scheme that does not build the program's control flow graph. Finally, our link-time allocation transforms the high and low-level IRs of the MIRV compiler instead of working at a binary level.

The Sparc architecture's register windows are a hybrid register/memory architecture intended to optimize function calls [Tami83, Weav94]. Each subroutine gets a new window of registers, with some overlap between adjacent register windows for the passing of function arguments.

Because Wall's link-time allocator splits the registers across active functions, it is a software approximation of the behavior of register windows [Wall88]. Wall concludes that using profile information to guide the link-time allocator always produces better performance than register windows. Variable-sized windows are slightly better than fixed-size windows, and it is more important to add a set of registers for global variables than adding more registers for the window buffer. This points out the importance of allocating global variables to registers.

Chow describes a similar interprocedural approach to register allocation where the available functions are visited in depth-first order and allocation made from the bottom of the call graph towards the top [Chow88]. He calls this the "depth first interprocedural register allocator." In the ideal case, if function A calls function B, then B will have been allocated before A is visited. Then when A is being allocated, it can see the registers that are in use by B and avoid using them. This allows the compiler to omit prologue and epilogue code to manage callee save registers. In effect, this pushes the callee save spills upwards in the call graph, eliminating them from the functions closer to the leaves. Like Wall's allocator, this approximates the behavior of hardware register windows.

3.8.3. Register Promotion

Cooper and Lu examined promotion over loop regions. Their algorithm is most similar to what is presented here [Coop98b], though our alias analysis is a form of MOD/REF analysis which is somewhat simpler than used in any of the previous work. Sastry and Ju examine promotion over arbitrary program intervals using an SSA representation [Sast98]. Lo and Chow use a variant of partial redundancy elimination to remove unnecessary loads and stores over any program region [Lo98].

All of the previous work shows substantial reductions in the number of dynamic load instructions executed and varying reduction in the number of stores eliminated. Typically 0-25% of loads are removed and up to 40% of stores are removed, depending on the program. Explicit load/store instructions are needed for register promotion because the global variables share registers which are local in scope. Cooper and Lu's results indicate that the main benefit of promotion comes from removing store operations. The other previous work shows that loads are improved more than stores. This disparity is primarily because the baseline compiler optimizations are not reported in any detail in any of the papers, which makes it difficult to perform a fair comparison. Two of the papers only counted the improvement compared to the total number of *scalar* load and store instructions [Sast98, Lo98]. While this shows the improvement of one aspect of the memory bottleneck, it does not show how effective promotion is at removing overall memory operations. One of the papers did not consider the effect of spilling because it simulated with an infinite symbolic register set before register allocation [Lo98]. Spill code is inserted by the allocator if too much promotion occurs in a region. Each of these papers reported on the improvement in dynamic load and store counts after their promotion algorithm was applied. They did not report on results from a cycle simulator as we do here.

3.9. Conclusions

This work first demonstrated the lack of consensus regarding register set size in research and commercial products. Research has been contradictory, ranging from statements like "compiler technology cannot utilize a large register file," to the other extreme that register sets of 100 or more registers are necessary for best performance. For commercial processors, register set sizes from 8 to 128 are seen in both the embedded and high performance realms.

While this is the case, the majority of high-volume processors have 32 or fewer registers. This can be attributed to a number of causes, primarily related to human programming style and the variables that are considered as candidates for allocation.

We have demonstrated in this chapter that existing advanced compiler optimizations can easily make use of 64 or more registers in general purpose codes. A reasonably aggressive optimizing compiler can make use of registers by allocating global variables, promoting aliased variables (in regions where each is accessed through one name only), as well as inlining, to improve performance 0% to 25%. Global variable allocation alone can improve performance up to 20% in some cases, and is the most effective of any of the optimizations at reducing execution time and memory operations. The combination of optimizations reduced cache operations by 10% to 60%.

These performance improvements were possible even when the number of cache misses did not decrease appreciably when optimizations were applied. This demonstrates two additional important conclusions. First, the allocation techniques used here are effectively moving data from the data cache into the register file. Second, these optimizations eliminate cache accesses, effectively speeding up access to data that would otherwise reside in the cache. Since cache hits are the common case, this is an important optimization.

This chapter also demonstrated that we can use even more registers if the compiler can allocate registers such that active functions share the register file. In this case, 100 to 200 registers are necessary to satisfy the register requirements of most benchmarks. We did some additional experiments with register windows and found that the combination of additional local and global registers (best112) with the optimizations presented here and register windowing allowed significant execution-time reductions over the original baseline binaries. Many benchmarks have 40% to 50% fewer memory references after all of these optimizations. In compress, 63% of memory operations are eliminated. Many of the benchmarks improve by 40% or more in cycle count.

The results are clear. A large register file is a necessary component for high performance. It is just as important to have optimizations in the compiler to take advantage of these registers, however. So it could be said that “aggressive compiler optimizations require a large register set, but a large register set requires aggressive compiler optimizations. It is also important to consider values that live across function call sites. We also found that cache misses are not the only component of performance that needs to be optimized: cache hits, since they are the common case, must be optimized as well. Other techniques that reduce cache miss penalty such as prefetching, data layout, and smarter cache replacement policies, are orthogonal to the work here.

We note that there are other candidates for register allocation that we have not explored in this work. Aliased data items cannot reside in registers for their *entire* lifetime or in regions where they are aliased unless support is added to the hardware. We will examine this in Chapter 5.

Chapter 4
Register Caching

4.1. Introduction

A large logical register file is important to allow effective compiler transformations or to provide a windowed space of registers to allow fast function calls. Unfortunately, a large logical register file can be slow, particularly in the context of a wide-issue processor which requires an even larger physical register file, and many read and write ports. Previous work has suggested that a register cache can be used to address this problem. This chapter proposes a new register caching mechanism, which takes a number of good features from previous approaches, whereby existing out-of-order processor hardware can be used to implement a register cache for a large logical register file. It does so by separating the logical register file from the physical register file and using a modified form of register renaming to make the cache easy to implement. The physical register file in this configuration contains fewer entries than the logical register file and is designed so that the physical register file acts as a cache for the logical register file, which is the backing store. The tag information in this caching technique is kept in the register alias table and the physical register file. It is found that the caching mechanism improves IPC up to 20% over an un-cached large logical register file and has performance near to that of a logical register file that is both large and fast.

4.2. Background and Motivation

The previous chapter showed that a large logical register file is a very important aspect of an instruction set architecture because it allows significant opportunity for compiler optimizations. Such optimizations have been shown to eliminate memory operations and speed program execution. Specifically, a logical register file of 64 or more entries is desirable to house locals, optimization temporaries, and global variables [Post00b]. Recent commercial architectures have underscored the importance of a large logical register file as well [IA6499]. More logical registers can also enhance ILP by eliminating memory cache operations, thus freeing the cache for more critical memory operations. The elimination of memory instruction reduces instruction fetch and decode time. Other techniques such as register windowing require a large logical register file in order to eliminate spill/reload traffic [Patt81].

Unfortunately, any large register file with many read and write ports is not practically implementable at clock speeds which are marketable, even if the performance advantage of the large file is compelling at slower clock speeds. There have been a number of proposals to circumvent this problem for large register files: either by physically splitting the register file, or by providing a cache of the most frequently used registers and having a large backing store for the full *logical* set of registers [Gwen96, Yung95a, Yung95b, Swen88]. The primary observation that these caching proposals rely on is that register values have temporal and spatial locality. This is the same principle that makes memory caches work.

A slightly different observation drives this work. Locality in the rename register reference stream in a typical out-of-order microprocessor is different than in the logical register reference stream, because renaming turns the incoming instructions into a single-assignment-like program. No register is written more than once; this means that a write breaks the locality for a given architected name. Instead, the observation that we rely on for this work is that most register values are produced and then consumed shortly thereafter. This was mentioned in Chapter 2: around 50% of values are used so quickly that they can be obtained from a bypass path instead of from the register file. Of the values not obtained from a bypass path, many of them are available within the instruction window, i.e. within the speculative storage of the processor. Such values rarely even need to be committed to the architected state, simply because they will never be used again. Such commits are termed “useless” by previous work, which reports that this phenomenon occurs for 90-95% of values [Loza95].

These facts suggested that we investigate how to implement a large logical register file efficiently in the context of a superscalar processor–a problem that previous work does not specifically address. The results of our investigation is the subject of this chapter. The logical register file that we want to implement – 256 registers – is so large that it is actually larger than most rename storage in aggressive superscalar processors built today. The rename storage is responsible for maintaining the storage of speculative values as well as mappings that determine where the architected state of the processor can be found. This immediately suggested that we view the implementation’s smaller rename storage as a cache for the larger architected register file, which would act as a backing store. The point of this design is to achieve the software performance of a large and fast architected register file without having to pay the hardware cost of implementing it. The cache is meant to reduce the performance hit of the large logical register file.

Previous work has suggested several alternative approaches to designing register caches, but we will propose a new one in this chapter.

Before continuing with the explanation of our implementation, we must review two other pieces of background information: *register renaming* and what we call *register architecture*. Section 4.3 and Section 4.4 then describe the physical register cache mechanism in more detail. Section 4.5 enumerates several of the advantages and disadvantages of the proposal. Section 4.6 evaluates the proposed mechanism by comparing it to a lower and upper performance bound which have no register caching. Section 4.7 compares our work to previous work in the area of register caching, and Section 4.8 concludes.

4.2.1. Register Renaming

In out-of-order superscalar processors, register renaming decouples the logical register file of the instruction set architecture from the implementation of the processor chip. The instruction set may have 32 registers while the microarchitecture implements 80 “rename registers” in order to allow it to exploit instruction-level parallelism by simultaneous examination of a large window of instructions which have been transformed into a single-assignment language to remove anti-dependencies and output dependencies. These rename registers contain state which is speculative (because of speculated branches, loads, etc.).

Register renaming is implemented in several different ways in commercial microprocessors. These designs are surveyed in detail elsewhere [Sima00] but we describe them briefly here.

One mechanism is called the merged register file, used in the MIPS R10000 and Alpha 21264 processors [Moud93, Yeag96]. In this design, the architected state and rename state are mingled in a single large register file which we will call the physical register file. Both speculative and non-speculative state share the same storage structure in this design. The register renaming and register release mechanisms must be designed so that architected state is maintained in a precise way.

The second implementation of register renaming is the split register file. The architected state is kept separate from the speculative state; each have their own register file and are updated appropriately. This approach is used in the PowerPC 6XX and PA 8000 processors.

The third approach is similar to second in that the architected state is separate from the speculative state, but the speculative state is stored in the reorder buffer. This technique is used in the P6 (Pentium II and III) microarchitecture.

Though renaming decouples the rename storage from the logical view of the architecture, the merged file approach is constrained in that it must implement more rename storage than there are logical registers. We denote this condition by saying that NPR > NLR must hold. Here, NPR is the number of physical (rename) storage locations, and NLR is the number of logical registers in the instruction set architecture[1]. This constraint is most simply explained by noting that the rename storage must have enough registers to contain all of the architected state plus some number of registers to support speculative execution (the result of running ahead of the architected state using branch prediction, etc.). Thus, the merged file approach does not take advantage of a complete decoupling of the logical storage from the rename storage.

Another way to explain this problem follows. The merged file approach renames each incoming instruction with a new physical register obtained from a *free list*. In order to reclaim a physical register in such a design requires several conditions to be met: 1) the value has been written to the physical register; 2) all instructions that require that value

[1] We use the notation NPR and NLR throughout the remainder of this chapter.

have a copy of it; 3) the physical register has been unmapped, i.e. the value has been superseded by later architected state. These conditions were enumerated in essentially the same form in earlier work [Moud93] and forms the basis of the MIPS R10000 microarchitecture [Yeag96].

One way to ensure these conditions are met is to allow a physical register P to be freed and reclaimed when the logical register L that it maps has been written by a later instruction which has also been committed to architected state. This condition guarantees that there can be no future consumer instructions that use the old value of L. We call this condition the *free-at-remap-commit* condition since the register can be freed when a later mapping of it is committed.

The processor could release P earlier if it knew that there would be no future consumer instructions which need the value. This would require perfect future knowledge of the instructions coming into the processor. Furthermore, since precise interrupts are required in most systems, it is generally required that the processor be able to stop execution between any two instructions, storing all of the architected state away so that later when the program restarts, it can resume with the register values as before. Since interrupts are not predictable at runtime, no processor can have this oracle knowledge, even for a small window of the program's execution.

The conditions enumerated above are the constraints placed on a merged-file renaming technique which are necessary to enforce correct dataflow in the program's execution. The particular implementation to address this constraint (free-at-remap-commit) has a deadlock condition if the number of physical registers in the merged file approach is less than or equal to the number of logical registers (NPR <= NLR). We call this the *register-release deadlock*. Suppose a machine with 2 physical registers, 2 logical registers, and a 3-instruction sequence which writes r1, then writes r2, and then writes r1. When the first instruction is fetched and renamed, r1 is assigned p1. Similarly r2 is assigned p2. Now when the third instruction is fetched, there are no physical registers remaining, so it waits until one becomes available before it proceeds. Eventually one of the first two instructions will be completed (e.g. the first one). At this point, however, we do not know if condition (2) is satisfied because there could be future instructions that want to use the value of r1. The machine cannot free p1 until it is assured of that condition. The only way this could happen is if another instruction that writes r1 is fetched and renamed. But we already noted that the fetch and rename stages are halted until a register is free. Thus the two portions of the pipeline are waiting on each other and neither can proceed. The essential problem is that forward progress depends upon instructions that have not yet been seen by the processor.

This difficulty constrains the logical file to be smaller than the physical file, a condition contrary to our initial desire, and prompted us to consider a design which allows more complete decoupling of the physical storage from the logical storage by splitting the logical and physical value storage instead of merging them. The next subsection explains another consideration that makes this conclusion attractive.

4.2.2. Register Architecture

By the term *register architecture*, we mean the number and configuration of registers need to support program execution. The register architecture has two facets – one facet is the logical register architecture, i.e. the number and configuration of registers supported in the instruction set. The other facet is the physical register architecture, i.e. the number and configuration of registers in the implementation.

The logical register file should be as large as desired by the software that runs on the machine. We have shown earlier that compiler optimizations have a significant impact on the logical register architecture, and that it should generally have more registers than most architectures today.

The number of storage locations in the implementation, on the other hand, is related to implementation technology, design complexity, and desired machine capacity, factors which are decided long after the instruction set has been fixed. These factors are mainly related to the physical capacity of the machine, including the instruction window size and number of function units and their pipeline depths. The physical register file must be matched to the characteristics of the design in order for the design to be balanced, instead of being matched to the instruction set architecture.

These different requirements mean that a strict decoupling of their designs is advantageous to allow the designer to maximize the performance of both the compiler and hardware implementations.

We consider the split register file model of register renaming. This decision was made based on the three factors just described: 1) the logical file is larger than the physical register file and thus has a natural backing-store to cache relationship; 2) a merged approach to register renaming cannot have more logical registers than physical registers because of the register-release deadlock problem; 3) the design of the logical file should be decoupled from the design of the physical storage to allow the designer the most freedom to optimize each individually. Other advantages of this selection will be presented later in the chapter.

4.2.3. The Physical Register Cache

The new register caching technique that is introduced in this chapter allows a large logical register file to be implemented at a realistic cost. It integrates a number of out-of-order processor hardware elements and combines a number of features of previous designs to arrive at a novel solution to the problem of building a large logical register file. The generic model is shown in Figure 4.1. It consists of the physical register file (PRF), which contains speculative results and some non-speculative results (this is the rename storage) and a logical register file (LRF) which contains precise architected state at all times. The PRF will have as many registers and ports as required in order to have a balanced execution engine; the LRF will have as many ports as can be sustained within the desired cycle time. Note that though the LRF may be much larger, it’s access time may not be much worse than the PRF because it will have fewer read and write ports. The other terminology that we will use is described in Table 4.1.

Instruction values committed by the processor to architected state are committed from the final stage in the pipeline. This is to avoid having to read the value out of the physical register file at commit and is why there is no direct path in the diagram from the PRF to the LRF. Alternatively, read ports could be added to the PRF to allow committed values to be read from it and sent to the LRF.

By design, the PRF is smaller than the LRF to allow for a fast cycle time. The PRF caches recently computed results, and maintains those values as long as possible. The cache contains the values most recently defined by the processor. In this way, the archi-

Acronym/Term	Meaning
LRF	Logical (Architected) Register File
NLR	Number of Logical (Architected) Registers
PRF	Physical Register File (the cache)
NPR	Number of Physical Registers
RAT	Register Alias Table; maps logical registers to virtual registers
VRN	Virtual Register Number
NVR	Number of Virtual Registers
PRFV	Physical Register Free Vector
Architected State	Committed, in-order, non-speculative state of the processor, visible at the instruction set architecture interface.

Table 4.1: Summary of terminology

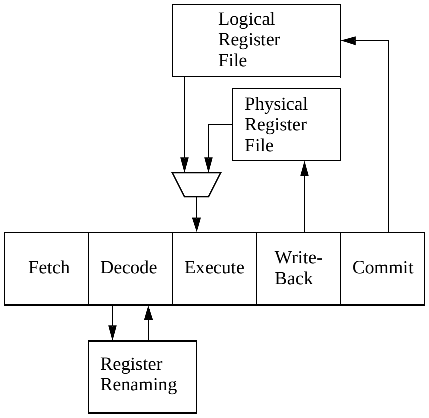


Figure 4.1: The machine model considered in this chapter.

tected file can be large and somewhat slower while the smaller physical file can have many ports to supply operands to the function units quickly.

The logical registers are mapped to the physical registers through a third (larger) set of “registers” called the virtual register numbers (VRNs). There is no storage associated with these VRNs, which are used to avoid the register-release deadlock, to allow the PRF to be directly indexed instead of associatively indexed, and to allow the PRF (cache) to maintain the values after they are committed to the LRF.

4.3. The Physical Register Cache Design

The innovation in this chapter is the combination of four mechanisms: separate logical and physical register files, a physical register file that is smaller than the logical file, renaming through a larger set of virtual registers, and a simple indexing scheme that maps the virtual numbers to physical registers. The combination of these techniques are used to achieve several goals: to provide a large logical register file that does not impact the speed of the processor’s critical path, to avoid deadlock conditions in register assignment that are problems in previous work, and to provide an efficient mapping from virtual number to physical register.

4.3.1. Microarchitecture Components

The major components in the microarchitecture are as follows:

1. A large logical register file (LRF) which contains precise architected state at all times. There is storage associated with logical registers: there are NLR entries in the LRF. Values are written to the logical register file at the time an instruction commits to architected state.
2. A set of virtual register numbers (VRNs). There is no storage associated with virtual registers: they are just numbers that track data dependences and the location of data (physical or logical register file) [Gonz97]. There are NVR virtual registers, where NVR > NLR. The virtual registers are assigned such that the low bits of the identifier index directly into the physical register file and the remaining high bits are called the “check tag.” The purpose of the check tag will be explained later. A VRN is allocated and deallocated as in the merged renaming approach, described in Section 4.2.1.
3. A physical register file (PRF) which contains speculative values eventually destined for the logical register file. The PRF also has value storage: it has NPR <= NLR entries. Values are written to the physical register file after an instruction computes its result. A result register’s value is retained after its producing instruction is committed to architected state, until a new value overwrites it. The PRF is directly indexed, unlike in some previous work [Cruz00].

The physical register file also contains tag bits (from the virtual register number check tag) to verify that the requested value is present; otherwise the value can be found in the logical register file because it was already committed and overwritten by a later producer instruction.

4. A virtual number free list (VNFL) which contains the numbers of all virtual registers that are currently available for use.
5. A physical register free vector (PRFV) of NPR entries where each bit represents whether the physical register is free to be allocated.
6. A rename table (RAT) of NLR entries each of which contains the virtual register number for the corresponding logical register.
7. A busy bit table of NLR entries which contains a bit for each logical register indicating whether it is presently being written by an instruction in the pipeline. If the bit is clear, then the value can be found in the logical register file. If set, there is a producer instruction in the pipeline which will produce the value at some future point.
8. A set of reservation stations. Each contains the following fields, assuming two source operands for convenience of explanation: a) dest virtual register; b) src1 ready bit; c) src1 source virtual register; d) src1 logical register number; e) src2 fields as for src1.

4.3.2. Design Constraints

We have identified several constraints that must be met by the design.

1. NPR < NLR. This is the basic assumption of the cache design of the NPR.
2. NLR < NVR. This ensures no deadlock condition in renaming since the rename register set is larger than the logical register set [Moud93].
3. Number of In-flight Instructions <= NPR. We limit the number of instructions to no more than the number of physical registers. This ensures that each instruction has a unique slot in the physical register file for its result. No two uncommitted instructions can have the same physical index. In other words, the number of instructions in flight cannot exceed the machine’s capacity.

Parameter	Value	Comment
NLR	256	The instruction set architecture allows 8 bits for each register specifier.
NPR	64	From the machine capacity.
NVR	512	Since NVR > NLR must be true (constraint 2 above) we specify 9 bits of virtual register number. The low 6 bits of this are used to index into the 64 physical registers. The remaining 3 bits are used as the check tag.

Table 4.2: Physical register cache parameters used in this study.

4.4. Operation of the Physical Register Cache

This section describes the detailed operation of the caching mechanism we proposed. For this study, we chose the parameters shown in Table 4.2.

When an instruction arrives at the dispatch stage, the source register operands are renamed based on the contents of the RAT, as in conventional renaming. Both the virtual register number from the RAT and the logical register identifier are carried with the instruction into the reservation station. No values are read from any register file at this time. In this way, values are stored centrally in either the logical or physical register file and are not duplicated in the reservation station entries.

Each destination register in the instruction is assigned a virtual register number from the free list, as in conventional register renaming. There is one difference: the physical register free vector is also queried. No virtual register number whose (six) physical index bits are currently in use can be chosen for allocation. This additional constraint is necessary to ensure that no two instructions share a physical index and is a necessary side-effect of the simple mapping policy that is used to index the physical register file (described later). This works without deadlock since the number of virtual registers is larger than the number of logical registers. Once a physical register number meeting this constraint is chosen, its free bit in the PRFV is cleared to indicate that this physical register has been “pre-allocated”. A register that is pre-allocated is marked as being in use as a destination register. Our scheme allows the value currently in that register to still be used

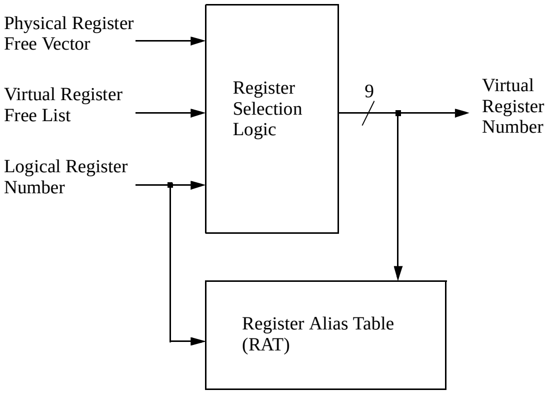

Physical Register Free Vector
Virtual Register Free List
Logical Register Number
Register Selection Logic
9
Virtual Register Number
Register Alias Table (RAT)

Figure 4.2: The mechanism for renaming a destination register.

by consumer instructions until the value is over-written. The relevant structures are overviewed in Figure 4.2.

Two bookkeeping structures are shown in the figure to allow the processor to select an appropriate register. The physical register free vector knows about all the free physical storage locations in the system. Similarly, the virtual register free list does the same for virtual registers. An autonomous register selection circuit can examine this information and determine which virtual-physical pairs are available for allocation, and can put them onto a third list which the processor then pulls from in order to rename the destination of an incoming instruction. (This list is not shown in Figure 4.2, but is inside the register selection logic box.) Essentially, the circuit is looking for a virtual register whose six index bits describe a free physical register. In our system, there are 64 physical registers and 512 virtual tags, so that for any physical register there are 8 possible virtual registers that can meet this criterion. The register selection circuit tries to find pairings where both are free. The register selection circuit has flexibility to choose the registers according to any policy that it likes and can effect different caching policies “offline”. If there is no virtual register that qualifies for renaming, the frontend of the processor stalls until one becomes available.

The newly renamed instruction is then dispatched and waits in a reservation station until its operands become ready. Readiness is determined when a producer instruction completes and broadcasts its virtual register number to the reservation stations. Each station compares its unready source VRN with the virtual register number broadcasted. If there is a match, the source is marked ready.

At some point all the instruction's operands are ready and it is scheduled (selected) for execution. The low 6 bits of its source operand VRN are used to directly index into the 64-entry physical register file. This simple indexing scheme constrains the initial selection of the VRN (in the renaming pipeline stage) but greatly simplifies the register access at this point. No associative search is necessary.

The upper 3 bits of the VRN are used as the 3-bit check tag whose function is to verify that the value currently in the physical register comes from the correct producer. If the PRF entry has a matching 3-bit check tag, then the value in the physical register is taken as the source operand. If the tag does not match, the value no longer resides in the PRF (like a cache miss) and must be fetched from the LRF. This means that it was committed to architected state some time ago and was evicted from the physical register set by some other instruction with the same low 6 bits in its virtual number. In the case where the value is not available from the physical register file, an extra penalty is incurred during which the backing store (logical register file) is accessed. Our indexing scheme does not allocate back into the cache upon a miss because the value that would be allocated no longer has a VRN (it was committed).[2]

When the instruction issues to a function unit, it picks up the necessary source operands, some from the LRF and some from the PRF. The LRF access can be started in parallel with the PRF access, if there are enough ports on the LRF to support this. This is shown in Figure 4.3, though this approach would require as many ports on the LRF as on the PRF. Alternatively, the LRF can be accessed the cycle after it is determined that the PRF did not contain the value. This latter approach is used in our simulations. Since each physical register could be accessed by multiple consumers in a single cycle, multiple ports are required on the PRF, but this is no different than other register file designs. Special scheduling logic, such as exists on the Alpha 21264 to handle remote register reads, is necessary in our system to handle the timing when a cache miss occurs.

[2] It is interesting to note that the no-reallocation policy makes our scheme look somewhat like a generational garbage collection system for registers. The short-lived items are in the PRF; the long-lived items are in the LRF and never move back to the short-lived structure.

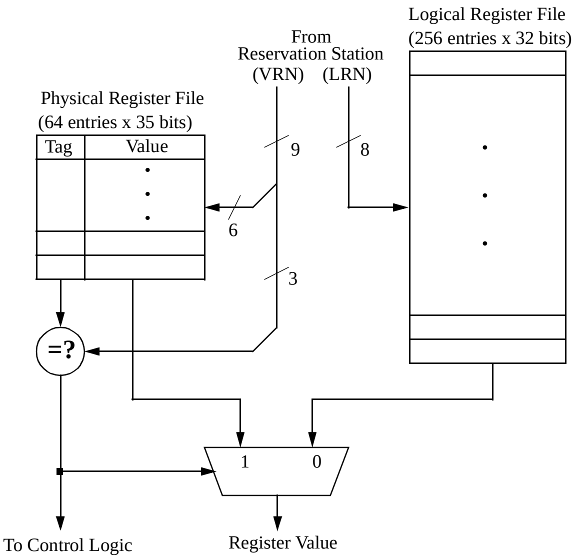


Figure 4.3: Mechanism to access a register value from the cache or backing store.

Immediately upon completion of execution, the (speculative) data is written to the physical register file. It is written to the index specified by the destination virtual register number. The check tag at that location in the PRF is also updated with the 3-bit check tag from the current instruction's virtual register number. This completes the allocation of the physical register for the current instruction. Any previous value that happened to be there is now overwritten and its value must be accessed from the LRF. We ensure that we do not overwrite a value which has not been committed yet because we require that no other in-flight instruction share the same 6 bits in its virtual register number (by the way the tags were selected). A write to the PRF always "write-allocates" its result this way and never misses the cache because it is a first-time write of a speculative value. It cannot be written to the LRF until it is proven to be on the correct execution path.

The instruction then broadcasts its VRN to each reservation station entry to indicate that the value is ready so that other instructions can be readied for execution (as in conventional systems). The physical register file is updated and the result is forwarded immediately to any consumers that require it. The virtual register number allocation algorithm ensures that the instruction retains its physical register at least until it commits to architected state.

Finally, the result of the instruction is carried down the pipeline into the reorder buffer. If this were not done, then the PRF would need more read ports in order to read out values at the time they are committed to the LRF.

When the instruction reaches the head of the instruction window and is able to commit, its value is written to the LRF (architected state) and the instruction is officially committed. The physical register is marked as free for use by later instructions. This is done by resetting the bit in the physical register free vector. The value in the physical register file, however, remains until it is absolutely necessary to overwrite it. This means that later consumers can read from the physical register for some time until the physical register is allocated to some other instruction. This eviction scheme is not simply “least recently defined” but is instead determined by when the next instruction that needs that register completes execution.

Virtual register numbers are released in the same manner as rename registers are released in a conventional, R10K style processor [Yeag96]. The virtual register number for logical register R1, for example, can be released when another virtual register number is assigned to R1 (at the next definition of R1) and that definition is committed. Virtual register numbers have no associated storage, so we can specify as many as needed in order to avoid stalling issue due to lack of them. No early release mechanism is considered.

Because the LRF maintains precise architected register state between each instruction in the program, recovery from exceptions and branch mispredictions is simple. Instructions which are younger than the excepting instruction are cleared from the machine. Older instructions are retained. The logical to virtual mappings are maintained as in previous work [Yeag96]. No entries from the physical register file need to be destroyed because any consumers that would have consumed bogus values have been cleared from the machine, and the bogus values will just be overwritten at some future point anyway[3]. This has the advantage that useful values are retained in the physical file (even those that have already committed); i.e. the cache is not destroyed even for a mispredicted branch. Were this not the case, then the LRF would have to supply all values

[3] Any garbage values in the physical register file will be ignored by later instructions since they will either find their (committed) values in the LRF or receive them from the result broadcast mechanism.

initially after a branch misprediction; this would be slow because the LRF will not have very many ports.

4.5. Physical Register Cache Advantages and Disadvantages

This design has many advantages, some of which are found in previous work but integrated here into one system. These advantages all arise from the main features of the design, namely the split LRF and PRF, the large set of virtual register numbers, and the way virtual numbers are mapped to physical registers using a simple indexing scheme (which pre-allocates physical registers). The advantages will be discussed in these categories.

4.5.1. Advantages Due to the Split LRF and PRF

The split design allows the LRF to be large while keeping the physical file small. This is the key to the cache-like behavior of the PRF. At any point in time the LRF contains precise architected state, making state maintenance easy, particular at points of misspeculation or other exceptional events. The logical register file needs fewer ports because it only supplies values that have been committed to architected state and that do not still reside in the cache. It need not provide values that are supplied from the bypass paths nor those from the (smaller and faster) PRF.

The split design also extends naturally to a register windowed architecture where there can be a large number of logical registers (hundreds) but where the PRF is desired to be smaller to speed execution. This is feasible since only a subset of the logical registers (one or two windows, say), are ever active at any one time.

4.5.2. Advantages Due to the Use of Virtual Register Numbers

The mapping of logical register to physical register through the virtual register numbers has a number of inherent advantages. First, the approach avoids deadlock conditions that would exist in a merged register logical/physical file where NLR >= NPR. The "up-rename" from logical to virtual number has no deadlock problem. The subsequent "down-rename" does not have a deadlock problem because the split physical and logical register files allow physical registers to be freed as soon as their values are committed, rather than waiting for future instructions to enter the machine.

The use of VRNs also mean that dependency tracking is separated from physical value storage. Virtual numbers are used to track dependencies while a separate PRF contains the values. This advantage was first proposed in previous work [Gonz97, Gonz98, Monr99]. The PRF is sized according to the capacity of the machine, independent of the size of the architected register file.

Virtual registers can be allocated in any order, and the order that is selected can implement a caching policy by keeping certain values in the physical register file longer after they commit than other values. This means that a trade-off can be made between machine capacity and value caching. In other words, some physical registers can be tied down to specific logical values so that reads can be satisfied out of the cache. This reduces the number of physical registers available for renaming but the increased PRF hit rate may more than outweigh this problem, given that mispredictions and other non-idealities effectively reduce the exploitable window size anyway. Such a trade-off would obviously be more applicable to a machine with a large number of physical registers, i.e. a machine that can more easily afford such a reduction in physical registers.

4.5.3. Advantages Due to the Direct Virtual to Physical Mapping

The simple mapping scheme from virtual to physical register number also has a number of advantages. Extra cycles are not required to access the physical register file. Previous work has resorted to using a fully-associative physical register file, which we believe is not feasible [Cruz00]. The only disadvantage of this approach is that the physical register selection mechanism in the renaming pipeline stage is somewhat complicated since it needs to make sure that it assigns a virtual register whose physical register is free. Thus it needs to make a lookup in both the virtual register free list and the physical register free list.

Additionally, physical registers are pre-allocated as soon as the instruction enters the machine to avoid complex mechanisms to steal registers or reserve registers when machine capacity is overrun. These mechanisms were necessary in previous work to avoid deadlock conditions when a physical register was required but not available [Gonz97, Gonz98, Monr99].

Pre-allocation of physical registers is performed without overwriting the previous value assigned to that physical register. Actual allocation is not performed until the instruction completes and writes back to the physical register file. This provides the opportunity for older values to hang around in the cache, satisfying consuming instructions faster than had the value been forced to reside exclusively in the LRF after its own commit. The late allocation feature of our approach also reduces the pressure on the physical file.

Physical registers can be freed as soon as the value contained is committed to architected state, unlike in other renaming configurations where a later write to the same logical register is required in order to free the register (like in the free-at-remap-commit mechanism). Releasing a physical register is decoupled from the number of consumer instructions in the processor because a physical register can be freed and the consumer can still access the value from the LRF. Previous techniques hang on to the physical registers longer than this [Moud93, Gonz97] and thus could reduce the number of in-flight instructions because of lack of available registers.

Even though the physical registers can be freed early, the PRF can retain committed values until they are overwritten. This means that, for example, branch mispredictions can be handled nicely because values produced before the mispredict could still be valid. Otherwise, if the PRF contents were destroyed at a misprediction, the LRF would have to provide all the register bandwidth by itself. This would be contrary to our desire to have few ports on that large file or else would slow the machine’s misprediction recovery time.

4.5.4. Other Advantages

There are some advantages which are due to the particular design that we have selected, but these are not inherent in this proposal. Because operands are fetched from the physical and/or logical register files at instruction issue (when sent to a function unit), data values need not be buffered in the reservation stations; the reservation stations are thus smaller and require less wiring. Similarly, data values need not be forwarded to reservation stations but are either sent to the physical file or via forwarding bus to an instruction that will consume it immediately.

4.5.5. Disadvantages

There are a number of features of the design which could be problematic. We list them in this section.

1. The number of write ports on the LRF must match the commit bandwidth of the machine. Conceptually, this is necessary because architected state must be capable of keeping up with the rate of instruction graduation in order for the machine to be of a balanced design. It does not need to keep up with function unit production however, assuming that some instructions are wasted because of mispredictions. One way to deal with this is to split the logical file into two (or more) pieces and to direct writes to either piece based on bits in the register specifier.
2. The RAT has as many entries as the (large) LRF. It must have enough read and write ports to keep pace with instruction decode. Each entry is only 9 bits wide and the structure is direct mapped, which somewhat simplifies the task of making it fast. This cost is due to the requirement of a large LRF. For future work, it may be profitable to examine ways of implementing the rename tables for a large LRF.
3. Our scheme always writes the value to the physical register file upon completion instead of selectively caching it as in previous work [Cruz00]. Values cannot be written to the backing store (LRF) until the instruction is committed, and we do not consider any way to avoid caching values which are used on the bypass network.
4. Before instruction execution can complete, the 3-bit physical register check tag must be compared against the 3 upper bits in the source VRN specifier. If the bits match, then the value requested is the one actually residing in the PRF and execution can proceed uninterrupted. If there is not a match, then an extra lookup must be made into the LRF to get the value, which was previously committed. This adds extra complexity to the supply of operands to instructions,

but the check can be made during instruction execution, and if a miss is indicated, the instruction can be squashed and immediately reissued after the value is fetched from the LRF. If the hit rate is high enough, the second lookup will not have to occur frequently.

4.6. Experimental Evaluation

4.6.1. Experiment Setup

All the benchmarks used in this study were compiled with the MIRV C compiler. We ran variants of the SPEC training inputs in order to keep simulation time reasonable. A description of MIRV, our compilation methodology, and benchmark inputs is presented in the technical report of [Post00a].

All simulations were done using the SimpleScalar 3.0/PISA simulation toolset [Burg97]. We have modified the toolset (simulators, assembler, and disassembler) to support up to 256 registers. Registers 0-31 are used as defined in the MIPS System V ABI [SysV91] in order to maintain compatibility with pre-compiled libraries. Registers 32-255 are used either as additional registers for global variables or additional registers for local caller/callee save variables.

All simulations were run on a subset of the SPEC95 and SPEC2000 binaries compiled with inlining, register promotion, global variable register allocation, and other aggressive optimizations for a machine with 256 registers, half set aside for locals and half for globals. These extra optimizations are most effective in the kind of machine studied here–namely one with a large number of logical registers.

We have implemented our register caching scheme on a variant of the `sim-outorder` simulator. The rename logic is duplicated for the integer and floating point files, so each of the descriptions below applies for each. When an instruction uses both types of registers, we return the maximum access latency. The register cache simulator is a standalone module which is hooked in several places into the `sim-outorder` simulator. The following are the descriptions of the main hooks.

At instruction dispatch, each output operand is renamed by finding the first VRN on the free list whose corresponding PR is not marked in use. This is entered into the RAT and the PR is marked as in use. At instruction issue, the simulator determines if there is a miss in the physical register file (cache). It does this as follows: for each virtual input, it looks up its PR and determines whether the virtual tags match. If they do, the number of hits is incremented and no latency is added to the instruction. If there is a miss, the instruction is assessed another cycle to service the miss (we do not start the LRF access in the same cycle as the PRF access). At instruction writeback, the physical register is written along with its tag. This allocates the physical register, overwriting any old value. At instruction commit, each logical register output is written to the LRF and the PR is immediately freed. The previous VRN mapping for this LR is freed, if any.

Table 4.3 lists the register cache configurations used in our simulations. The latency numbers in the tables are the additional delay (on top of whatever delay is simulated in the `sim-outorder` simulator). All simulations use the register cache code with different parameters to simulate the various configurations of interest. The cached simulation is the one of interest in this work. It has a 256-entry LRF coupled with a 64-entry PRF. The LRF has an additional 1 cycle penalty for accessing it. The PRF has no additional penalty. The other three configurations (Fast, Slow1, and Slow2) are provided for comparison purposes. These are *simulated* with a 512-entry PRF so that the VRN to PR mapping is direct, with no check bits, and an appropriate latency. In effect, each of these three models always hits in the PRF and never needs to use the LRF. “Fast” is a model of a 256 logical register file machine which has no extra delay to access the registers. “Slow1” and “Slow2” are similar but add 1 and 2 extra cycles, respectively, to the register access to simulate the slower register file.

The remainder of the simulation parameters are common across all simulations. The machine simulated is a 4-issue superscalar with 16KB caches; the parameters are used from `sim-outorder` defaults except for two integer multipliers and 64 RUU entries. The default parameters are described in our technical report [Post00a]. All of the simulations presented in this work assume an infinite number of ports on each register file. This is necessary in order to allow observations to be made about the number of ports that would be desired in such a configuration.

Simulation Name	NLR	LR Latency	NVR	NPR	PR Latency
Fast	256	0	512	512	0
Cached	256	1	512	64	0
Slow1	256	0	512	512	1
Slow2	256	0	512	512	2

Table 4.3: Register cache simulation configurations

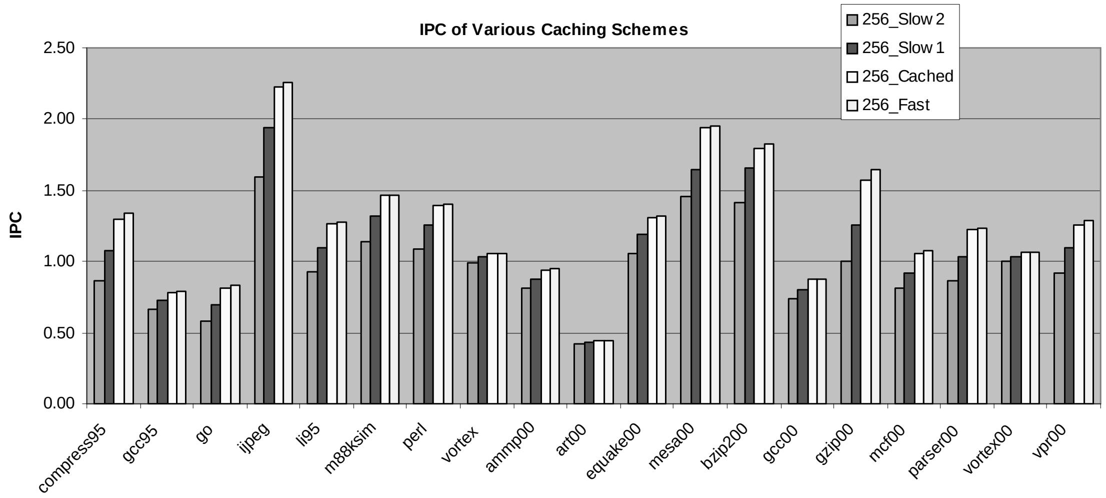


Figure 4.4: IPC of the four register configurations studied.

4.6.2. Results

This section describes the results of simulating our configurations. Figure 4.4 shows the IPC for the 4 register configurations mentioned above. The cached configuration approaches the performance of the fast configuration in most cases. The Fast configuration averages about 12% faster than the Slow1 while the cached configuration is 11% faster. Thus the caching strategy is able to recover most of the performance of the fastest configuration while maintaining a small physical register file.

Figure 4.5 shows the hit rate of the integer-side physical register file cache. In every case, the hit rate is 80% to 95%. Even the small cache is able to capture most of the register activity. This is due to effects that have been reported previously, namely that values are very frequently consumed very shortly after they are produced. Our simulator does not model value bypassing in detail, so the hit rate includes all values that would be

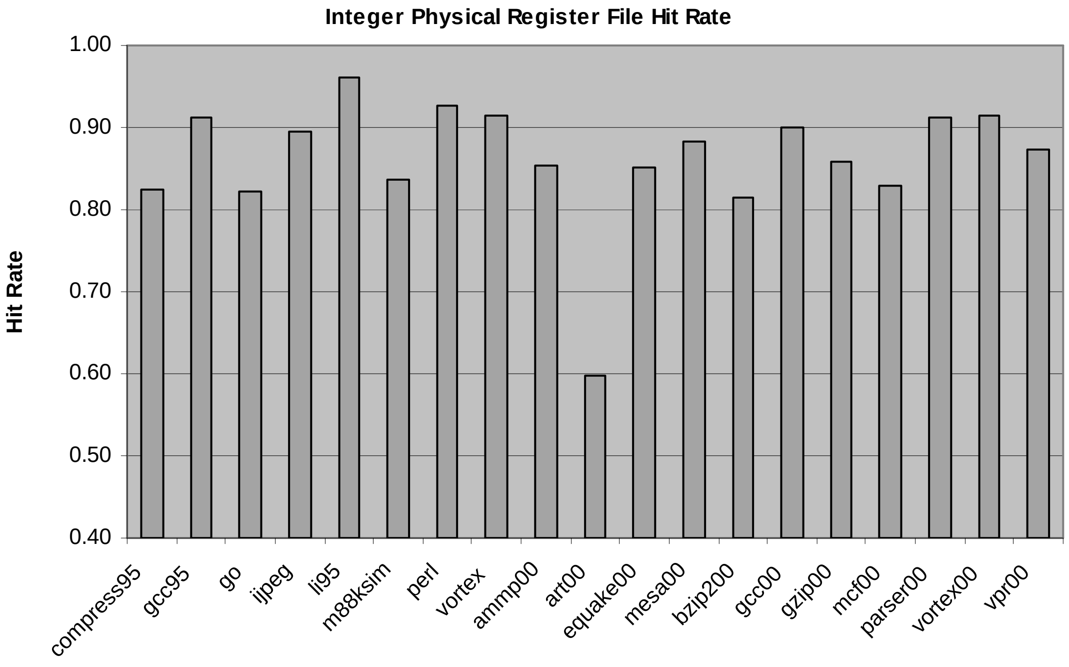


Figure 4.5: Hit rate of the integer physical register file (cache).

bypassed or provided by the physical register file. In any case, these are register references that do not have to access the large LRF, so we deem them to be “hits.” Even if 50 out of 100 values are provided by bypasses, 30 to 45 of the remaining 50 are provided by our cache–which is still a 60 to 90% hit rate. Similar data for 4 of the SPEC2000 floating point benchmarks are shown in Figure 4.6. The floating point hit rate is a bit higher, 85% to 95%.

Another important metric to examine is the average read and write bandwidth for the logical and physical register files. Figure 4.7 and Figure 4.8 show this data for the SPEC95 and SPEC2000 benchmarks, respectively. Each bandwidth number is an average which is computed by dividing the number of reads or writes to the particular register file by the number of cycles in the simulation run. This does not show peak port requirements.

Each benchmark has 4 bars, corresponding to the 4 configurations simulated (Slow2, Slow1, Cached, and Fast). The black and white portions of the bars represent average read and write ports needed to supply the bandwidth required from the PRF. Notice that since these configurations model a physical register file large enough to house

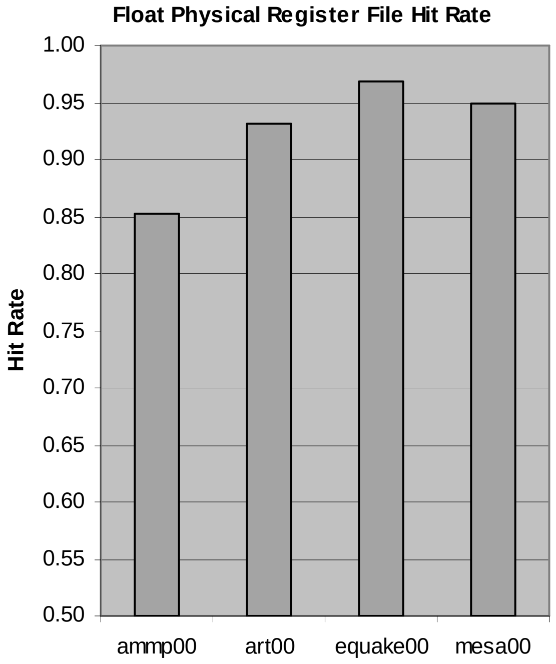


Figure 4.6: Hit rate of the floating point physical register file.

the architected registers and speculative state there are no reads or writes to a LRF. The PRF in these cases is 512 entries.

The third bar, for the Cached configuration, has two additional bars stacked on the graph: the gray and hashed bars represent the bandwidth requirements for the LRF. The key thing to note about the third bar is that the black and white port requirements are for a 64 entry physical register file–the other bars show the port requirements for a monolithic, 512-entry file. For most of the benchmarks, the caching technique retains about the same port requirements for the 64 entry PRF as for the larger monolithic file (just slightly less), indicating that the cache is supplying as many operands as the original file but with a fraction of the size. Thus the effect is that the cache is able to supply the needed operands with many fewer ports on a smaller register file.

The cached configurations show in dark gray the average bandwidth requirement of the logical register file for read accesses–a very small percentage, indicating that 1 or 2

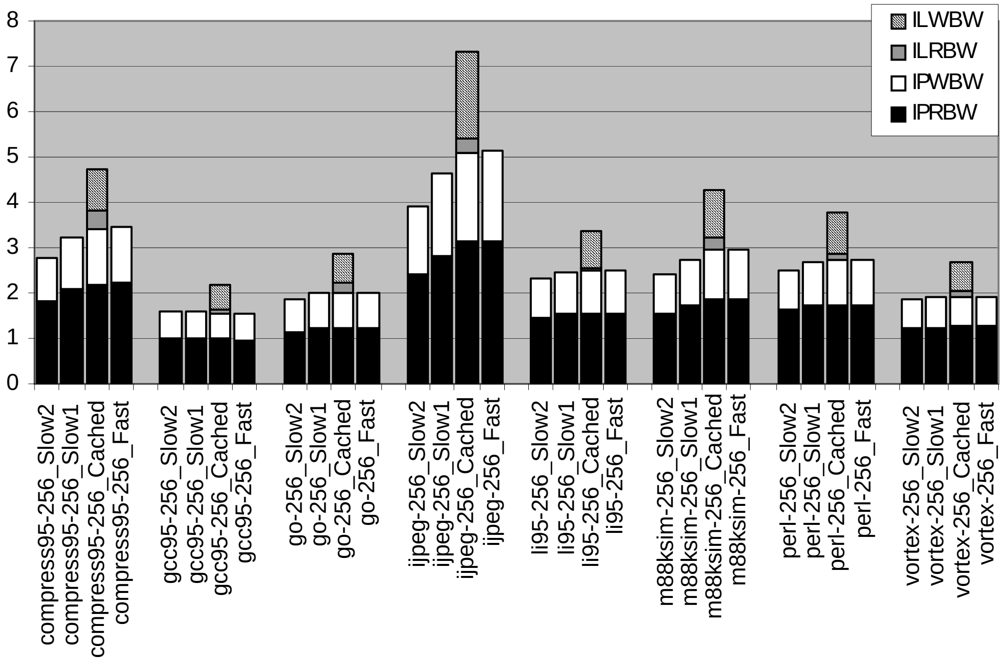


Figure 4.7: Average bandwidths SPECint95.

read ports to the LRF would be sufficient to sustain the performance levels in these benchmarks.

The data in those figures also show that the physical register file read bandwidth averages two to three ports per cycle and write bandwidth up to two ports per cycle. A physical register file of 64 entries with that number of ports should be quite easy to build with that few ports.

The logical register file write bandwidth, shown in the hashed portions of the bars, averages around one or two per cycle. The specific values are always less than the IPC of the benchmark, since some instructions like branches and stores do not write a value to the register file. The logical register file, as was previously noted, must have enough write ports to be able to keep up with the commit bandwidth of the machine. Since the IPC for these benchmarks is low, it should not present a difficulty to design a 256-entry LRF with two write ports.

Average bandwidth data is also shown in Figure 4.9 for a subset of the floating point benchmarks. The `mesa` benchmark is not shown because its floating point register utilization is very low compared to the other benchmarks.

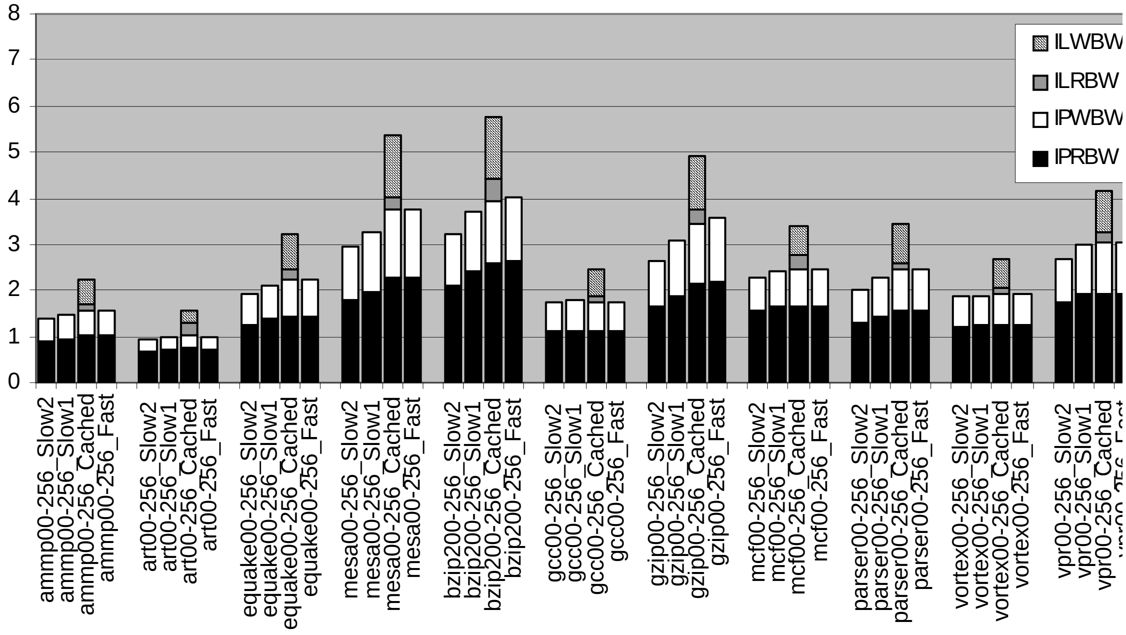


Figure 4.8: Average bandwidths for SPEC2000.

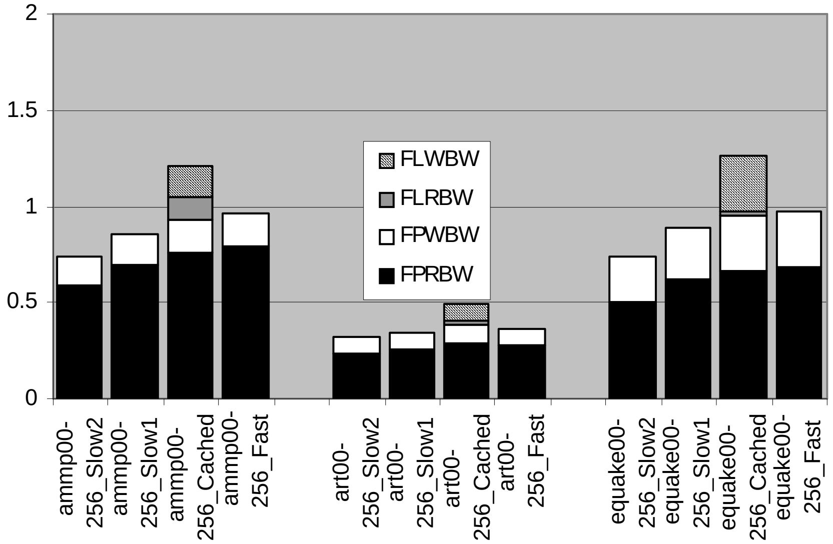


Figure 4.9: Average bandwidths for floating point register files.

The register cache allows values to hang around until they are overwritten by some later instruction; while the later instruction has pre-allocated the register, the old value remains in it. This is advantageous because the physical register is occupied with useful data for a longer period of time than if the value were “erased” as soon as it was committed to the LRF. All of the simulations presented so far use this policy, which we call “delayed allocation.” We simulated the go benchmark with and without this policy to model what would happen if registers were allocated in a manner similar to current superscalar processors. We found that when the delayed allocation was turned off, the hit rate of the integer PRF decreased by about 2.6%. There was also a slight decrease in IPC. This shows the delayed allocation policy is making a difference though it is very slight. The primary reason for this is that most values are consumed quickly and the extra time the value hangs around in the cache is not very profitable. However, it is easier to implement the delayed allocation policy, so the slight performance increase is worthwhile.

This modification extends the physical register busy time on the front end (before execution) because it makes the register busy until the value is produced, even though no useful value resides in that register. Turning off delayed allocation has increased the useless occupation time of the physical register. On the back end (after commit), our scheme releases the register as soon as the commit is completed. We cannot extend the register busy time until the commit of the next value writing the particular architected register because we would run out of physical registers (that is, as we said before, we cannot use a merged renaming approach like the R10000).

4.6.3. Results with Perfect Prediction, Caching, and TLBs

The absolute performance of the configurations in the previous section are somewhat attenuated by the heavy penalty of branch prediction and cache misses. This section removes those constraints by simulating all the configurations with perfect caches, perfect branch prediction, and perfect TLBs (denoted by the “perf” suffix on the configuration names in the following graphs). Our goal is to determine what happens to the caching scheme as improvements are made in other areas of the system. The figures in this section show the results.

Figure 4.10 shows the IPC of the four perfect configurations. Immediately it can be seen that the IPCs have increased from the 1-1.5 range up to the 2-3 range. The performance of go, for example, has tripled. More interesting than the absolute performance increases produced by the more ideal system is the trend in performance from the Slow2 to Fast configurations of the register cache. Since a number of the performance-attenuating features have been eliminated from the microarchitecture, the gap between Slow2 and Fast has increased. For example, whereas in the non-perfect simulations Fast was 44%

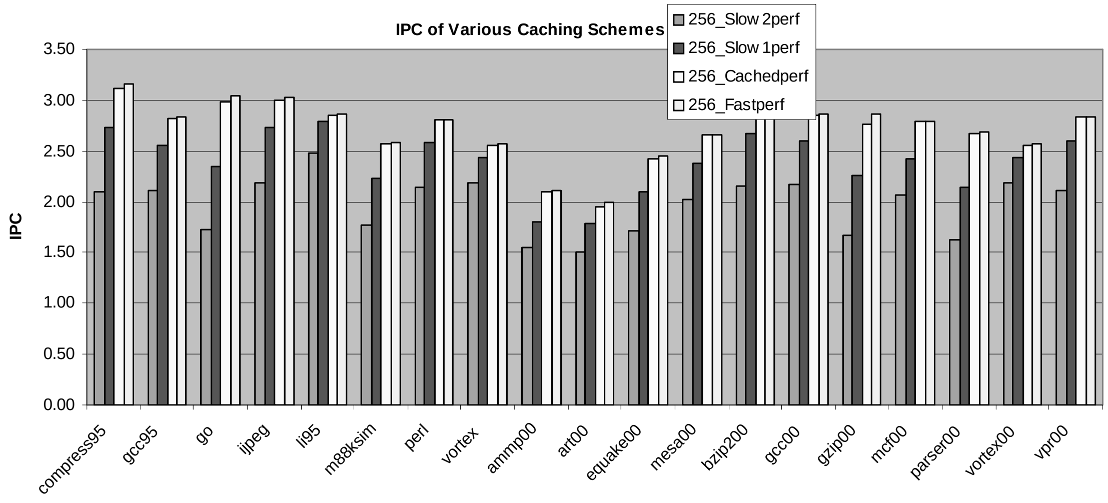


Figure 4.10: IPC of the four perfect register configurations studied

faster than Slow2, the perfect Fast configuration is 76% faster than Slow2. This points out the (somewhat obvious) conclusion that as the other bottlenecks are removed from the system, the register configuration makes a significant difference.

Figure 4.11 and Figure 4.12 show the hit rate of the integer and floating point physical register files for relevant benchmarks. These numbers are essentially unchanged from the non-perfect simulations, showing that the mechanism is robust under different configurations. The only significant difference is found in the `art` benchmark, where it attains a 47% hit rate as opposed to a 60% hit rate in the imperfect configuration.

There are also corresponding increases in the average read and write port bandwidths required. Figure 4.13 and Figure 4.14 show this for the integer register files; the port requirements on the physical file have increased to 4 read ports and 3 write ports. This is an increase of at least one port per cycle for reads and writes. The LRF port requirements on the integer side are still quite modest–generally less than 3 write ports and a single read port are required. The three floating point benchmarks, shown in Figure 4.15, require 2 to 4 read ports in the perfect configuration whereas the non-perfect configuration only required 1 read port. The PRF requires about 1 read port, as before, and the LRF requires one of each type of port also.

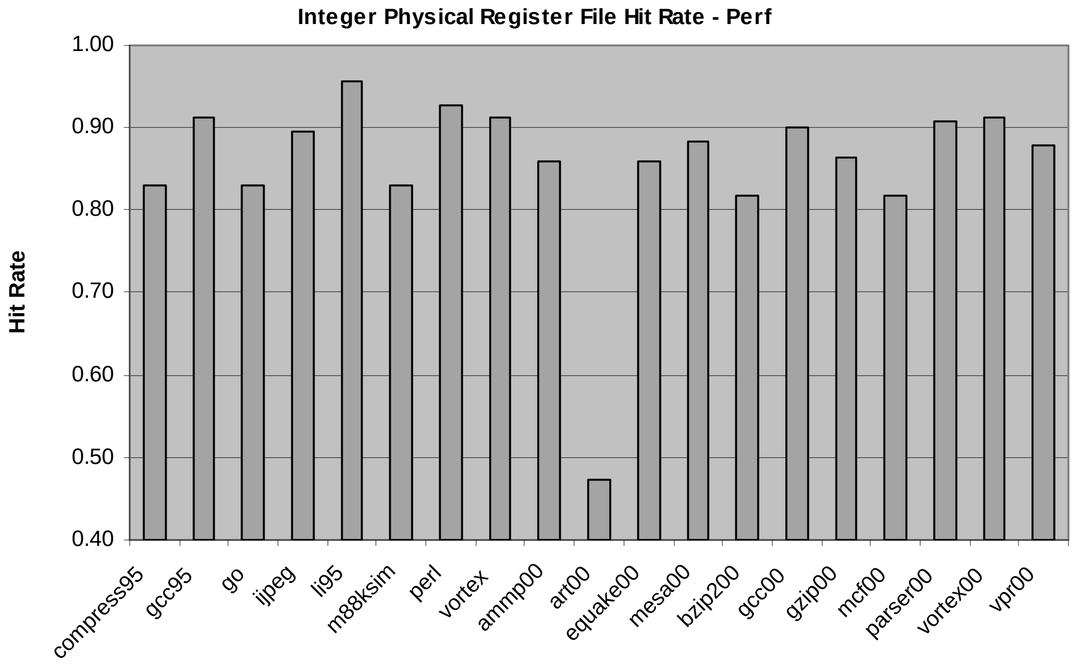


Figure 4.11: Hit rate of the integer physical register file (cache), perf configuration.

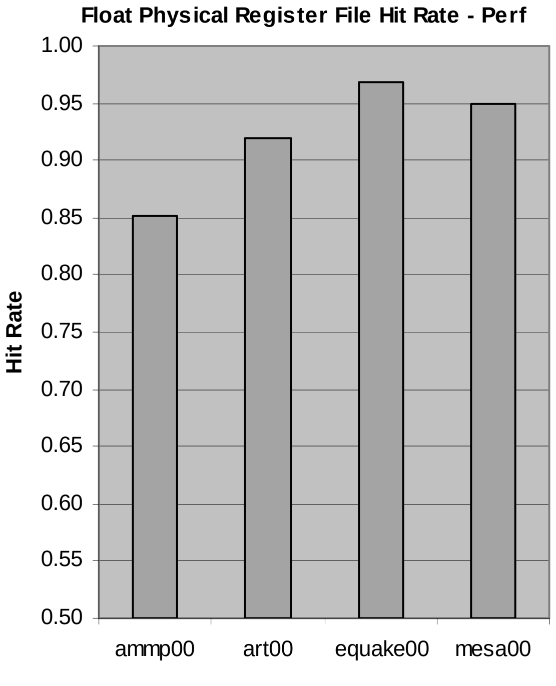


Figure 4.12: The hit rate of the float physical register file, perfect configuration.

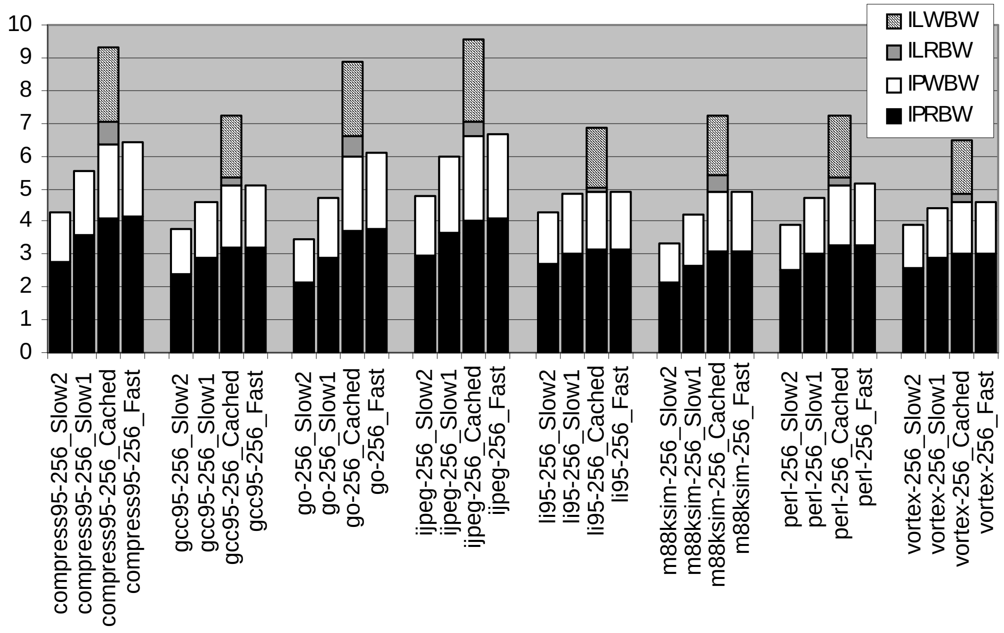


Figure 4.13: Average bandwidths SPECint95, perfect configuration.

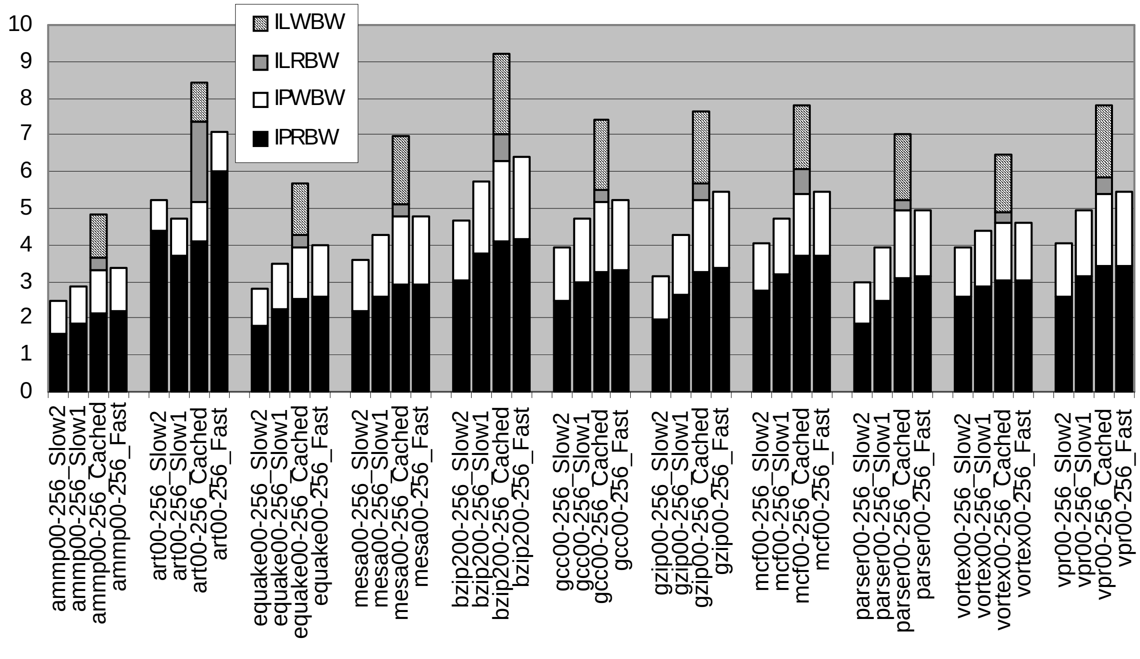


Figure 4.14: Average bandwidths for SPEC2000, perfect configuration.

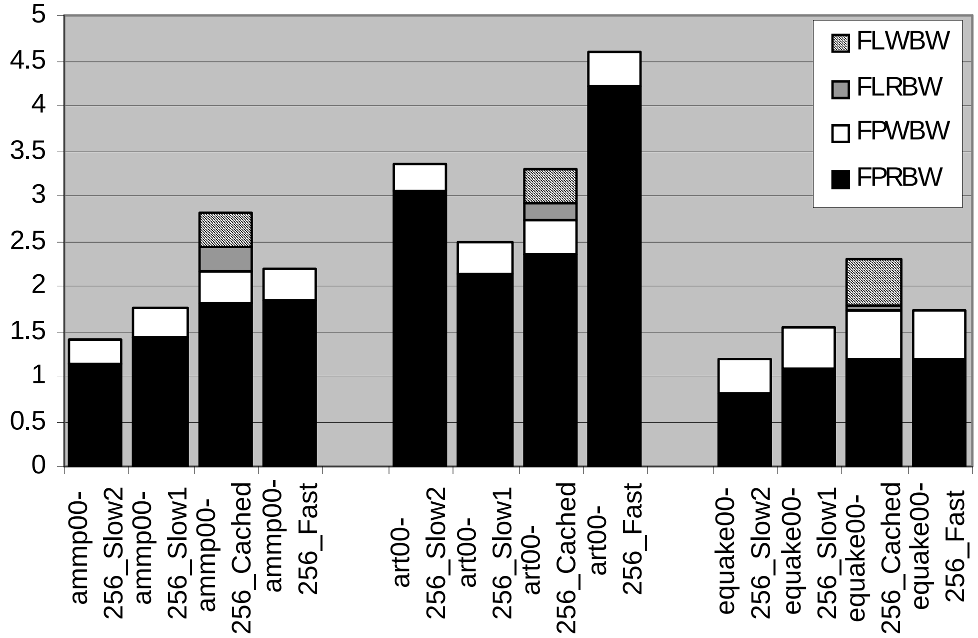


Figure 4.15: Average bandwidths for float register files, perfect configuration

4.6.4. Results of Varying Cache Size

We next discuss what happens when the cache size is altered. In our design, the PRF capacity is intimately related to the other parameters in the machine, such that changing it requires changing a number of other parameters as well. The cache size should be determined by the desired machine capacity and performance. For this reason, and because of the large number of simulations that would be required to examine a variety of cache sizes for all of our benchmarks, we have limited the discussion to the `go` benchmark. For each simulation, we changed the size of the cache (for the cached configurations) and reduced the size of the instruction window (for all the configurations) to match the capacity of the cache.

Figure 4.16 shows the IPC for the four configurations explained earlier, each simulated with five cache sizes: 8, 16, 32, 64, and 128 entries, and correspondingly-sized instruction windows. Some clear trends can be observed from this data. Generally speaking, the Fast configurations are always better than the Cached configurations which are in turn better than the Slow1, etc. However, there are a couple of exceptions. The cached8 configuration is slightly slower than the slow1-128, and the same as the slow1-64 config-

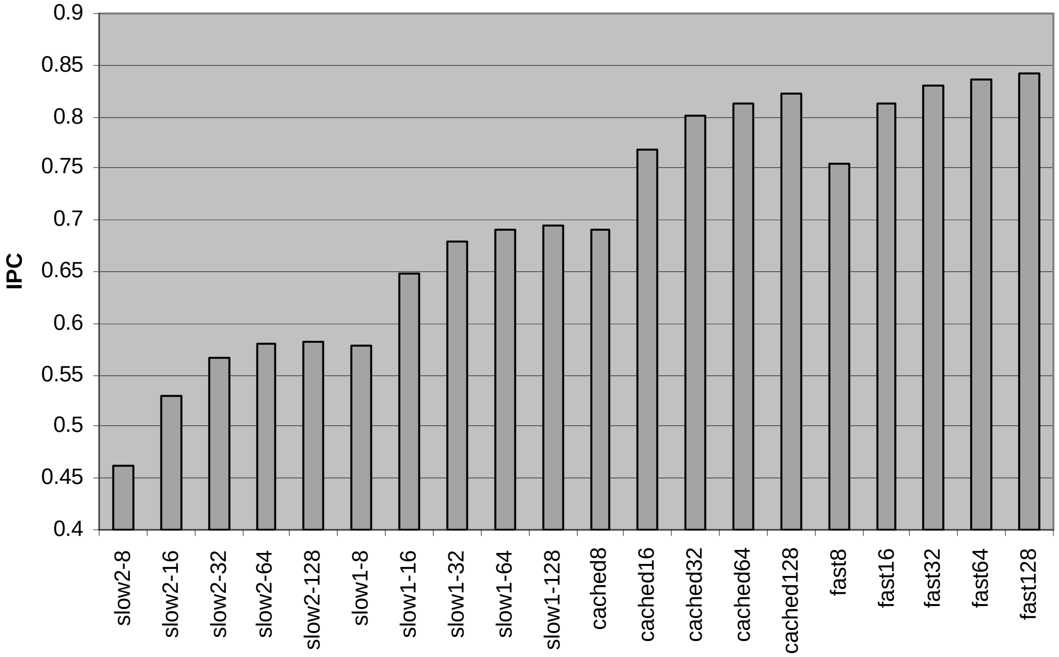


Figure 4.16: IPC for go for various cache configurations.

uration. This shows that more slow registers and a larger instruction window is better than too few. Similarly, the fast8 configuration is slower than all cached configurations except cached8. Fast16 is much better. The indication in both of these exceptional cases is that an 8-register cache and window size is simply insufficient to best performance on go. Of course, this is no surprise, but serves to make the point that changing the PRF size has drastic effects on the performance of the benchmarks.

Figure 4.17 shows the average register file bandwidths in each case, and the PRF hit rate is shown in Figure 4.18. The hit rate trends upward as the cache increases in size, from 65% up to 85%.

From the perspective of experimental design, this data does not tell us much because too many parameters in the machine are changed from one to the other, it is difficult to determine the effect that each has on overall performance. The tight integration of our caching model with the rest of the superscalar hardware makes it impossible to untangle these different parameters.

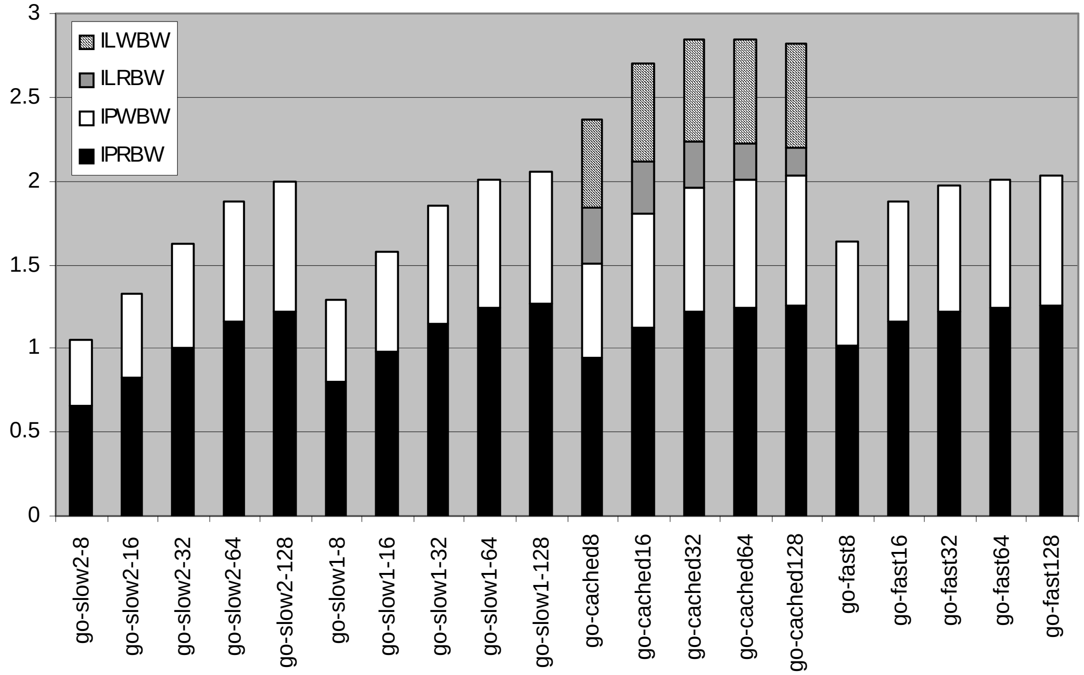


Figure 4.17: Average bandwidths for integer register files in go.

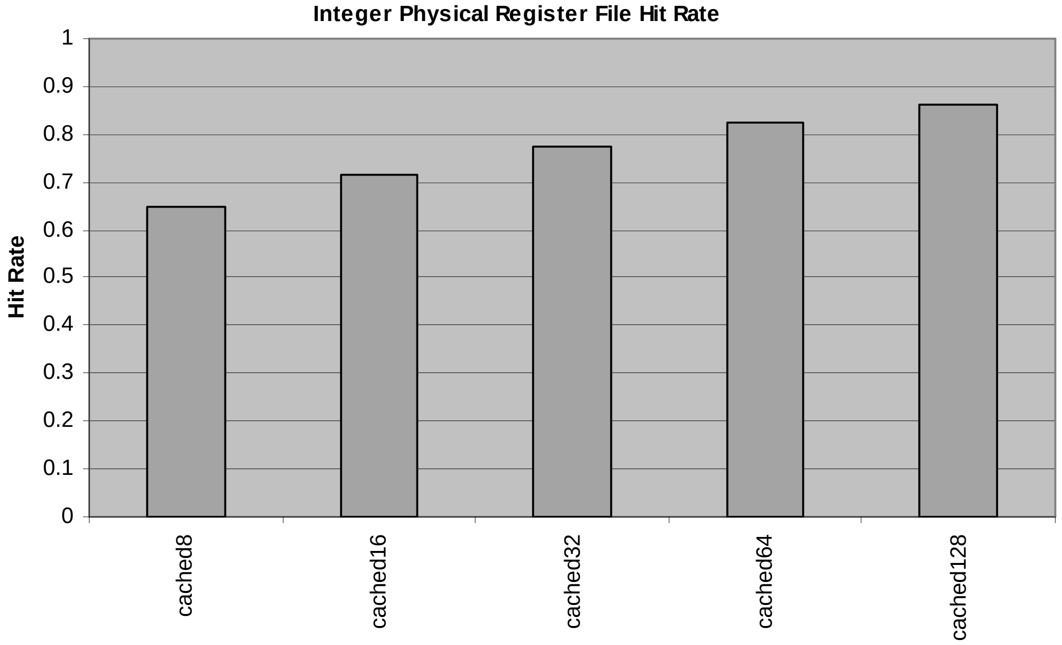


Figure 4.18: Hit rate of the various cache configurations.

4.6.5. Results of Varying Available Register Ports

The results above are only averages over the entire benchmark run and they assume an infinite number of ports to both register files. In this section we demonstrate what happens with several different configurations of ports on the register files.

For these simulations, the number of read and write port on each register file are used to restrict the flexibility of several portions of the pipeline model. The read ports of both the LRF and PRF guide the instruction scheduler so that it does not issue more instructions in a cycle than there are register ports to supply operands. The scheduler is optimistic in that it examines all ready instructions and starts selecting them for execution as long as the number of read ports from the appropriate register file is not exceeded. It continues through all ready instructions, perhaps skipping some that cannot be issued due to high port utilization, until it issues as many as it can, up to the issue width limit. The LRF write ports are used in the commit stage of the pipeline, where if an instruction cannot commit because of lack of LRF write ports, commit is stopped and the remainder of the retiring instructions must be committed the next cycle (or later). The PRF write ports are used in the writeback stage of the pipeline to write results from the function units. Our simulator assumes that the PRF must be able to sustain the writeback bandwidth of as many instructions that can complete per cycle. Therefore we do not restrict the PRF write ports[4].

There are minimum port requirements on each of the register files. These are shown in Table 4.4. Both the PRF and LRF must have at least three read ports each, since our simulator will only read the sources for an instruction in a single given cycle, and there are some instructions with three source operands[5]. This could be solved by taking several cycles to read the operands, but we deemed it not necessary to simulate fewer than 3 read ports since most PRF designs should have at least that many. Similarly, the simulator requires at least 2 write ports on each register file since some instructions have two desti-

[4] Though this is obviously an important issue in real machine design, we defer it for a more detailed simulation environment

[5] Fewer ports causes the simulator to lock up because once it reaches an instruction with more operands than ports, it cannot make further forward progress.

PRF Read Ports	PRF Write Ports	LRF Read Ports	LRF Write Ports
3	NA	3	2

Table 4.4: Minimum ports requirements on the PRF and LRF.

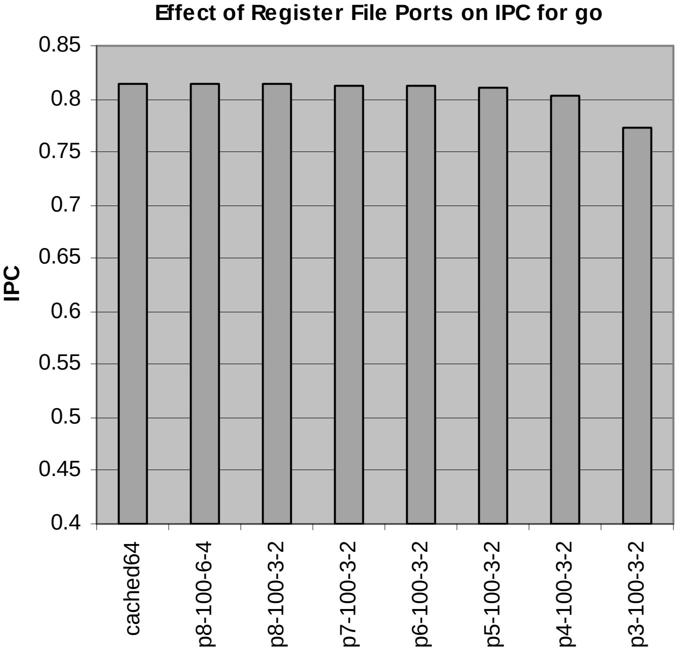


Figure 4.19: IPC of several limited port configurations on go.

nation registers. As mentioned above, we did not model limited write ports on the PRF, but we did on the LRF.

The results of several simulations are shown in Figure 4.19. The cached64 configuration is the same as the “cached” configuration earlier in this chapter. The next configurations are specified by the number of register file ports, in the same order as specified in Table 4.4, so that p8-100-6-4 means that the configuration has 8 PR read ports, 100 PR write ports (to model an infinite number of ports), 6 LR read ports, and 4 LR write ports. The next configuration halves the number of each type of port to the LRF, and the following configurations successively reduce the PR read ports from 8 to 3.

From the graph, it is evident that limiting the cache configuration from infinite ports to 8 PRF read ports and 6 LRF read ports and 4 LRF write ports does not affect the performance of the processor on the go benchmark at all. There is a slight degradation when the number of LRF read and write ports are halved, but this is not severe because go has such a low IPC anyway. When the number of read ports on the PRF is cut successively from 8 to 3, the performance drops by 5%. The performance is not affected until the number of read ports is reduced beyond 5. Thus, for the go benchmark, 5 read ports is sufficient to obtain nearly full performance from the PRF. Of course, this is due to the fact that go's performance is so low in the first place (less than 1 IPC).

Figure 4.20 and Figure 4.21 show the results for all of the benchmarks studied in this chapter. There is usually not much performance degradation going from infinite ports to the minimum number of ports on the LRF; however reducing the number of ports on the PRF to the minimum does affect performance. From the infinite port configuration to the most limited, performance is reduced 2% up to 24%. The ijpeg and bzip benchmarks perform the worst with the limited port configuration. This is not surprising since those two benchmarks have the highest average port requirements (see Figure 4.7 and Figure 4.8). The art benchmark produces the only unexpected result. This has been a difficult benchmark through all of the studies because it has such terrible overall performance. This is due to the very high data cache miss rates–the L1 data cache misses 42% of the time; the unified L2 cache misses 48% of the time. These misses cause major backups in the instruction window, so that it is full over 90% of the time. The most limited cache configuration slightly changes the order that instructions are run from the earlier configurations and thus it is not surprising that there is a small perturbation in the performance; in this case it is in the upward direction. The IPC is low enough that the PRF is easily able to sustain the average requirements (1/2 an instruction per cycle can easily be accommodated by 3 read ports on the PRF).

This data demonstrates that our technique is not hampered by a limited number of ports on the LRF. Primarily this is because data values are produced and then consumed shortly thereafter so that the cache or bypassing logic can supply the values. Furthermore, the out-of-order engine can tolerate the extra cycle incurred by a PRF miss. The PRF port limited studies show that performance does not really begin to degrade until the number of read ports is reduced to 5 or 4.

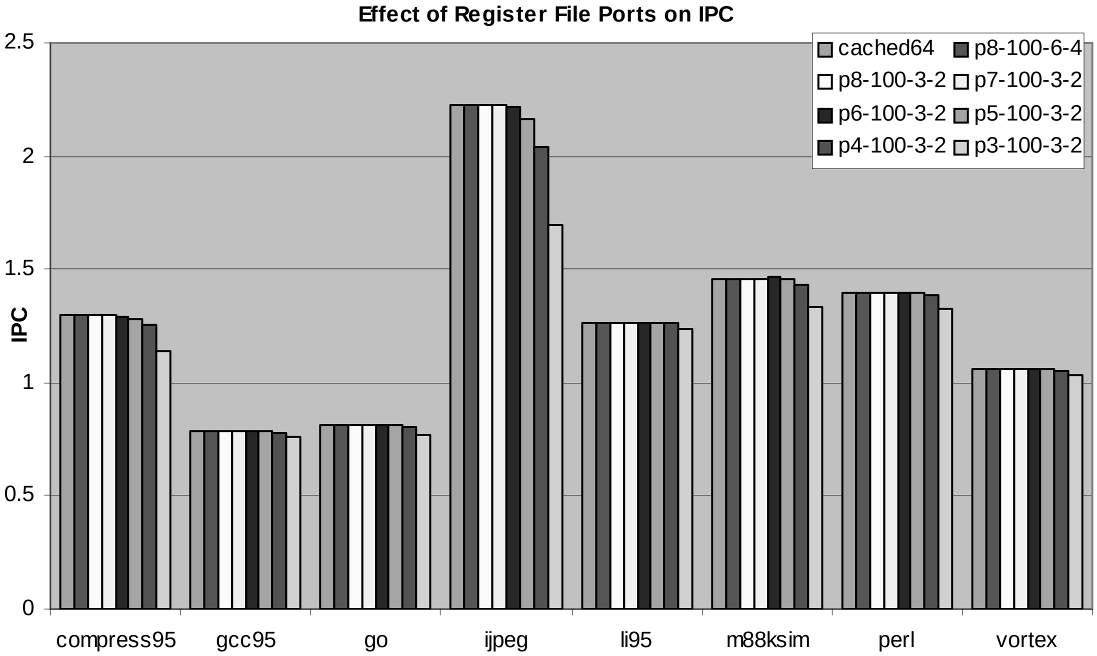


Figure 4.20: The IPC of limited port configurations for SPECint95.

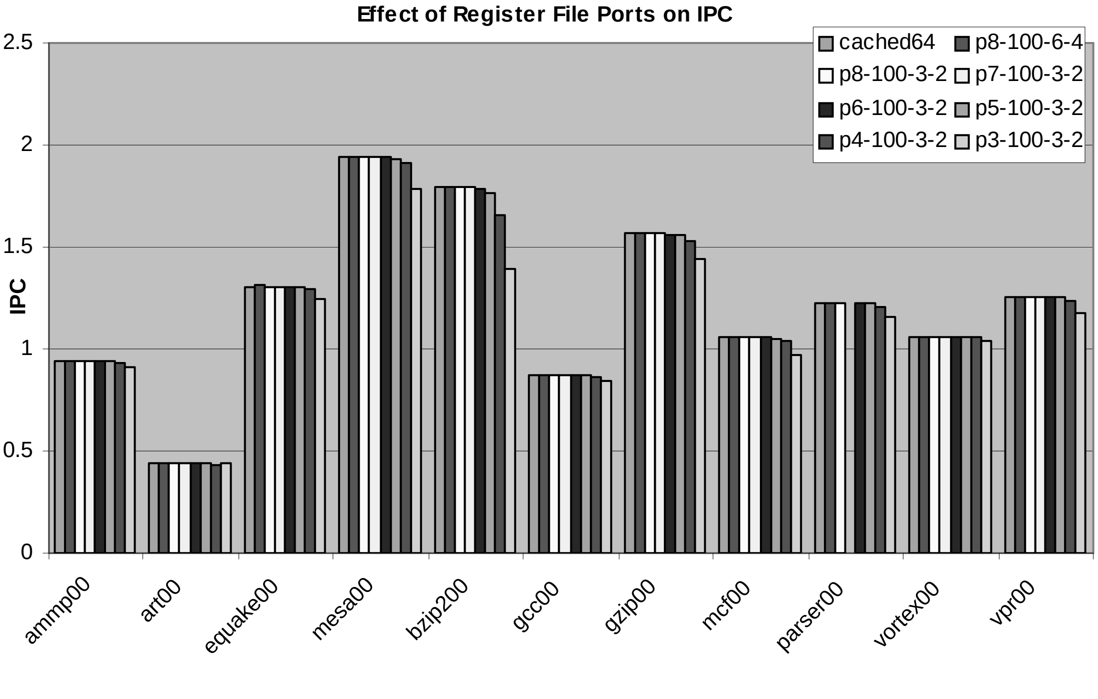


Figure 4.21: The IPC of limited port configurations for SPEC2000.

4.7. Comparison to Previous Work

The register cache is a hardware controlled mechanism for making using of temporal locality of register reference [Yung95a, Yung95b]. The register file is organized as a hierarchy with the operands supplied from the uppermost (smallest and fastest) level. The lower levels constitute backing storage for the full set of registers, not all of which will simultaneously fit into the small upper level. Motion between files is performed by the hardware based on recent usage patterns.

This strategy is used in a recent proposal, called the “multiple-banked register file,” where a mutli-level caching structure with special caching and prefetching policies is used to implement a large number of physical registers [Cruz00]. This work attempts to cache the physical registers of a dynamically renamed microprocessor in the context of a merged-file renaming mechanism, despite the seeming lack of locality in the physical register reference stream (because of the “random” selection of physical registers from the free list).

One disadvantage with the multiple-banked work and the present work is that the register file backing store (the logical file in our case, the large physical file in the previous work) must have as many write ports as the cache. In that work, all values produced by the function units are written to the physical register backing store; some of them are written to the cache as well based on caching policies. There is no dirty-writeback path from the cache to the backing store, so all values produced by the function units must be written to the backing store during instruction writeback. In our work, only committed values need to be written to the logical register file. This will require somewhat less bandwidth than the previous approach since the number of instructions committed is less than the number written back.

Another disadvantage of the multiple-banked research is that the physical register file, though small at 16 entries, requires a fully associative lookup on all ports. Our work eliminates this inefficiency by using a clever virtual-to-physical register indexing scheme to allow the physical file to be direct mapped.

Hierarchical register files have also been proposed to allow efficient implementation of large logical register sets. The register file is separated into several regions, each in turn containing more registers and having slower access time than the previous region [Swen88]. Placement of data is performed by the compiler with respect to frequency of access, and motion between files is explicitly coded by the compiler.

The problem of large register files has been addressed in commercial machines such as the Alpha 21264, which split its physical register file into two copies, with a 1-cycle delay for updates from one copy to another [Gwen96].

Other work attempts to reduce the number of physical registers required for a given instruction window size so that caching techniques will not be necessary. One example is the virtual-physical register research which makes use of the observation that physical register lifetimes do not begin until instruction completion, so that storage need not be allocated until late in the pipeline [Gonz97, Gonz98, Monr99]. In order to keep track of dependences, an intermediate level of register renaming, from logical registers to virtual-physical registers, is employed. The virtual-physical registers do not have associated storage so the number of them does not greatly affect the size or speed of the processor. These virtual-physical registers are renamed once again to final physical registers at instruction completion time. The delayed allocation of physical registers introduces a potential deadlock where there may not be a physical register available by the time an instruction commits, since the machine allows more instructions in flight than there are physical storage locations for them. This is corrected by reserving a number of physical registers for the oldest instructions and sending younger instructions back for later re-execution if they try to use one of the reserved registers. The technique was later changed to handle this deadlock better by implementing a physical register stealing approach [Monr99].

Our approach avoids this deadlock by separating the logical register file from the physical register file.

One advantage of the virtual-physical approach is that dependency checking is decoupled from value storage, thus allowing value storage to be tied up for a shorter length of time than in a standard superscalar. This is used to allow more in-flight instructions or to reduce the size of the physical register file while retaining the same performance. This allows late allocation of physical registers but early deallocation was not considered.

Our work is like the virtual-physical register approach in that we actually allocate the physical register at instruction writeback, though pre-allocation happens earlier. This proposal also frees the physical register as soon as the instruction commits, which is the earliest that any technique can do so. No merged file mechanism can do this because the value storage must be retained until it is certain that the value will never be needed again. Once the physical register is freed and the value is placed in the logical file, any future consumers can access it from there. Even though the register is freed, the value can sit in the physical register until some other writer destroys it. The check tag ensures that no later operations get a wrong value. This allows the value to sit in the cache for some time after commit. Some previous work has considered earlier deallocation of physical registers by using dead value information which exploits the fact that the last use of a register can be used for a deallocation marker instead of waiting for the next redefinition [Mart97, Lo99].

Figure 4.22 shows these differences in pictorial form. The clear bar represents regions where the physical register is allocated but does not contain a valid value. The black bar shows where the physical register is allocated and must contain valid data (until a new value is committed to architected state). The checked bar shows the region where the physical register is pre-allocated but does not contain data for the present instruction; it may contain valid data from an older instruction. Finally the shaded arrow represents the region where the physical register is free to be allocated to another instruction, yet it contains valid data from the previous producer instruction, which consumers are free to use. Therefore, the deallocate region overlaps the pre-allocate region for a later instruction that will use the same physical storage location.

Our work differs from previous research in that it proposes to use the physical register file itself as a cache for a large logical file using a new register-renaming technique. Previous work is mainly concerned with implementation of large physical register files whereas we are mainly interested in implementing a large logical register file.

The rename storage in previous superscalar designs could be considered as a cache for the logical register file. However, if the rename storage is larger than the architected storage, as it is in many modern superscalar processors, the “cache” is bigger than the

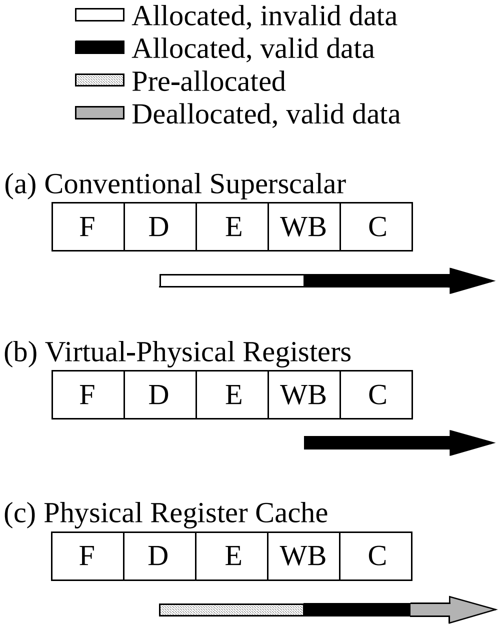


Figure 4.22: The lifetimes of physical registers in various schemes

backing store. In any case, our system is designed specifically to cache a large set of registers provided in the ISA to the compiler.

4.8. Conclusions

We have presented an implementation of a large and fast logical register file by integrating register renaming with a physical register file smaller than the logical one. This physical register file serves as a cache for the logical register file as well as storage for in-flight instructions. The renaming scheme is unique in several ways. First, the physical register file is smaller than the logical file. Second, the renaming scheme renames the logical registers to physical registers through an intermediate set of virtual registers. Third, the mapping function is constrained in such a way as to ensure that the physical register file is direct-mapped. This technique avoids the register-release deadlock problem and also deadlock problems of earlier virtual tagging schemes which had to reserve or steal registers to ensure that the program makes forward progress. The caching mechanism provides an improvement of up to 20% in IPC over an un-cached large logical register file with conventional register renaming. The hit rate of the physical register file on most benchmarks was 80% or better.

The caching mechanism proposed here can be extended in a number of directions. First, the proposal is amenable to modifications to effect caching policies on the physical registers through careful allocation of virtual registers. Second, the caching mechanism is a natural fit to be integrated with register windows for a SPARC-like architecture. Another way our approach could be used is to build an inexpensive superscalar implementation of a conventional (32-register) logical file with an even smaller number of physical registers, say 16, while using a direct-indexed physical file instead of the associative ROB lookups used in earlier designs.

Finally, this approach could be useful on simultaneous multi-threaded processors which require very large logical register files to house the contents of the multiple thread contexts that are simultaneously live in the machine. Previous research has used a merged register renaming scheme [Lo99], which means that the physical register file (which contains both architected and speculative state) must be extremely large. For example, for 4 threads at 32 registers each, the PRF would need to be larger than 128, and in particular it would be 128 plus the maximum number of in-flight instructions. Our technique could be used to implement a much smaller physical register file.

Chapter 5
Store-Load Address Table and Speculative Register Promotion

5.1. Introduction

Register promotion is an optimization that allocates a value to a register for a region of its lifetime where it is provably not aliased. Conventional compiler analysis cannot always prove that a value is free of aliases, and thus promotion cannot always be applied. This chapter proposes a new hardware structure, the store-load address table (SLAT), which watches both load and store instructions to see if they conflict with entries loaded into the SLAT by explicit software mapping instructions. One use of the SLAT is to allow values to be promoted to registers when they cannot be proven to be promotable by conventional compiler analysis. We call this new optimization speculative register promotion. Using this technique, a value can be promoted to a register and aliased loads and stores to that value's home memory location are caught and the proper fixup is performed. This chapter will: a) describe the SLAT hardware and software; b) demonstrate that conventional register promotion is often inhibited by static compiler analysis; c) describe the speculative register promotion optimization; and d) quantify the performance increases possible when a SLAT is used. Our results show that for certain benchmarks, up to 35% of loads and 15% of stores can potentially be eliminated by using the SLAT.

5.2. Background and Motivation

Register allocation is an important compiler optimization for high-performance computing. Access to data stored in machine registers avoids using the memory subsystem, which is generally much slower than the processor. Register promotion allows scalar values to be allocated to registers for regions of their lifetime where the compiler can prove that there are no aliases for the value [Coop97, Sast98, Lo98]. The value is *promoted* to a register for that region by a load instruction at the top of the region. When the region is finished, the value is *demoted* back to memory. The region can be either a loop or a function body in this work, though promotion can be performed on any program region. The benefit is that the value is loaded once at the start of the region and stored once at the end, and all other accesses to it during the region are from a register allocated to the value by the compiler.

Unfortunately, imprecise aliasing information and separate compilation conspire to limit the types and amount of data that can be safely allocated to registers. To allow a relaxation of the compiler's conservative nature, we introduce the *store-load address table (SLAT)* and investigate its use in enabling more effective register allocation. We also introduce a new compiler transformation called *speculative register promotion*, which makes use of the SLAT, and evaluate the performance gains it can provide.

The SLAT and speculative register promotion introduce several new opportunities for register allocation. Figure 5.1 shows the combinations that we consider in this chapter. Figure 5.1(a) is conventional register allocation as done by most compilers. Figure 5.1(b) shows the result of register promotion, which requires more sophisticated compiler alias analysis. (Throughout the chapter we use the term *alias* somewhat loosely to include all possible references to data though mechanisms other than its primary name, including ambiguous pointers and side-effects.) Figure 5.1(c) requires further compiler support because in order to prove that the global can be allocated to a register for its entire lifetime requires that the whole program be analyzed at once. This allows the compiler to make the determination that the variable `global` is only ever used through its name, and never through a pointer. Previous work has examined this optimization [Post00b, Wall86].

Figure 5.1(d) shows another example using default register allocation. This time the loop contains a function call, which means that conventional promotion (with separate compilation of functions) cannot be sure that `foo()` does not access the `global` variable. Thus `global` cannot be promoted to a register. Figure 5.1(e) shows how the SLAT allows promotion to occur anyway. The compiler promotes `global` as in normal register promotion but uses special opcodes to inform the hardware that the promotion is speculative. Finally, link-time global allocation can be done even under separate compilation

```
while () {
  ld r5, global
  add r5, r5, 1
  st global, r5
}
```

(a) Original source

```
ld r5, global
while () {
  add r5, r5, 1
}
```

(b) Register promotion

```
while () {
  add r32,
r32, 1
```

(c) Link-time global allocation

```
while () {
  ld r5, global
  add r5, r5, 1
  st global, r5
  foo();
```

(d) Original source

```
map r5, global
while () {
  add r5, r5, 1
  foo();
}
```

(e) SLAT-based promotion

```
while () {
 add r32, r32,
1
  foo();
}
```

(f) SLAT-based link-time global allocation

Figure 5.1: The results of using different register allocation strategies.
(a) The original source code, in a combination of C and assembler notation. It uses the default strategy for allocation, which does not allocate the global to a register. (b) Register promotion moves the load and store outside of the loop. (c) After application of link-time global variable allocation, each occurrence of global is replaced with r32 and unnecessary copies are removed. (d) Another snippet of source code, which includes a function call, rendering the global not promotable by conventional means. (e) The SLAT allows the promotion to occur in spite of the function call. (f) Link-time global variable allocation can also be performed with help from the SLAT even when separate compilation is used.

when the SLAT is used to protect the `global` variable. In this case, the mapping operation occurs at the start of the program–say at the top of `main()`–and is not shown in the figure. Table 5.1 gives a summary of these allocation strategies.

The remainder of this chapter is organized as follows. Section 5.3 describes the logical organization of the SLAT. Section 5.4 introduces the speculative register promotion transformation. In Section 5.5 we describe our experimental setup, while our experimental results are analyzed in Section 5.6. Section 5.7 describes previous work in the areas of memory disambiguation and register allocation. Finally, we discuss our conclusions and directions for future work in Section 5.8.

5.3. The Store-Load Address Table (SLAT)

The store-load address table (SLAT) is a hardware structure that allows the compiler to relax some of the conservative assumptions made due to imprecise analysis of memory communication. Logically, the SLAT is a table where each entry contains a logi-

Allocation Strategy	What is Allocated	Region in Register File	Is Whole-Program Information Used?
(a, d) Default	Unaliased local scalars including compiler temporaries.	Local	No.
(b) Register Promotion	Aliased local scalars or global scalars aliased or not. In either case, they are promoted for regions where they are provably unaliased.	Local	Can be used to enhance alias analysis so that extra candidates can be proven safe to promote.
(c) Link-time global allocation	Unaliased global scalars.	Global	Required.
(e) SLAT-based promotion	Aliased local scalars or global scalars. SLAT allows allocation even in aliased regions.	Mappable	Can be used to reduce number of SLAT promotions necessary.
(f) SLAT-based link-time global allocation	Aliased and unaliased global scalars.	Mappable	Can be used to reduce number of SLAT promotions necessary.

Table 5.1: Various strategies for allocating registers

In our usage, “aliased” means that the variable’s address has been taken somewhere in the program or it could be referenced through a function-call side effect. The register file regions are conceptual divisions of the registers into groups based on their function. The “local” region of the register file is the region used for local variables in the function. The global region contains global variables for their entire lifetime. The mappable region contains mapped (speculatively promoted variables). In our experiments, the local and mappable regions are the same. The letters in column 1 correspond to the labels in Figure 5.1.

cal register number, memory address and some information flags for bookkeeping. Speculative register promotion uses the SLAT to associate a memory address with a register. All references to this address will be forwarded to the register file as long as the address is mapped in the SLAT. Thus, the SLAT is indexed associatively by address.

Special machine instructions are used by the compiler to manage the SLAT. To initialize a speculative promotion, a special `map` instruction is used. This instruction includes a memory address and a register number. A SLAT entry is created, indicating that the data at the given memory address resides in the given register. A load from memory is also executed to place the desired data in the register. Likewise, an `unmap` instruction removes an association from the SLAT, sending the data in the register to the memory. The `map` and `unmap` operations are essentially just special load and store operations.

After a `map` instruction has associated a memory address with a register, every subsequent memory operation examines the SLAT, comparing its address operand with those in the SLAT. When a match (conflict) is detected in the SLAT, the memory operation is redirected to the register file. A load retrieves its value from the SLAT-mapped register instead of from memory; a store uses the mapped register as its destination instead of memory. An `unmap` instruction at the bottom of the promotion region handles storing the updated register out to memory.

Since the SLAT allows register allocation of potentially aliased variables (including globals that may be used by callee functions) whose scopes may exceed that of a single function, special handling is necessary to close the “gap” between function-scoped machine registers and registers containing mapped data. One example of this problem occurs at function call boundaries. On entry to the callee, all callee-save registers used by the function are first spilled to the stack to preserve existing values for the caller. These registers are restored upon function exit. If one of these callee-save registers is mapped in by the SLAT, the spill instruction must be dynamically modified to store the data to the “home” memory location of the data (the global storage or stack location for an aliased local variable). This home address is available in the SLAT entry for the register being spilled. A reload operation likewise must be modified to load from the home location. These operations require two new memory instructions: `spill` and `reload`. These are store and load instructions with special opcodes to indicate their function (saving and restoring of callee-save registers). These instructions must examine the SLAT to see if the referenced register is mapped. Thus the SLAT is also indexed directly by register number. We classify such registers as *callee-update*, analogous to callee-save, because their values are automatically updated by any memory accesses in the callee function.

Because the `reload` instruction must have access to the home memory address for the data, the processor must keep every SLAT entry that is created until an `unmap` deallocates it. Moreover, an address can be mapped to multiple registers or a single register can be re-mapped to a new address. These cases are simplified by the fact that only one mapping is active for a particular function. The compiler can guarantee that no address or register is mapped twice in the same region. It can do this because it only speculatively promotes directly-named scalar variables.

There are several strategies for dealing with these situations. One possibility is to have a large SLAT with a hardware-controlled overflow spill mechanism, similar to that used in the C-machine stack cache [Ditz82]. Another possibility is to require compiler management of the SLAT. Instructions to save and restore SLAT entries can be generated in the same way instructions to save and restore callee-save registers are generated. Our simulations assume an infinite-sized SLAT so that we may evaluate its performance potential.

In addition to callee-save spills and reloads, spill and reload operations are necessary to deal with excessive register pressure within a function. Speculative register promotion can increase the amount of this spilling. Since the spilling effectively negates the benefit of register promotion the compiler may simply reverse the promotion if spilling occurs. Memory access size and overlap must also be considered in the SLAT; the compiler can restrict promotions to ease this problem.

5.4. Speculative Register Promotion Using the SLAT

This section outlines how the SLAT can be used to allow speculative register promotion. Preliminary exploration into the limitations on static register promotion indicated that a significant number of memory operations cannot be promoted due to ambiguous or unseen memory accesses through function calls. This will be quantified later in the chapter. To address this problem, we consider a new optimization called *speculative register promotion* which uses the SLAT to allow promotions in these situations. It does this by providing a fallback mechanism in the case that the promotion was too aggressive, i.e. that there was a conflict where the promoted value was not synchronized with its value in memory. When this occurs, the hardware can provide the current value.

As we saw in Section 5.3, the SLAT is tailored to solve this problem because the hardware compares each load and store address against those stored in the SLAT. Once a value is promoted to a register with a `map` instruction, it can be used or defined several times before a conflicting memory load appears. Since the value in memory could be out of date with respect to the value promoted to the register, both load and store operations have to be examined to see if they are attempting to access the value that was promoted to a register.

The register promoter in our C compiler, MIRV, can promote global scalar variables, aliased local scalar variables, large constants, indirect pointer references (we call these *dereferences*), and direct and indirect structure references. It can do so over loops or whole functions. The algorithm is described in detail elsewhere [Post00b]. Speculative register promotion was a simple augmentation to the existing promoter. Any directly-named value (global or local) which is not promoted because of aliases can be promoted speculatively (based on simple selection heuristics). This is accomplished by emitting a promoting load (`map`) and demoting store (`unmap`) at the boundaries of the region, with additional information indicating these are speculative promotion operations. The backend of the compiler passes this through via annotation bits in the instruction encoding and the simulator treats the `map`/`unmap` operation as described in Table 5.2. Since global and aliased data can reside in registers, the compiler was also restricted from certain kinds of code motion around those accesses.

5.5. Experimental Setup

All the benchmarks used in this study were compiled with the MIRV C compiler. The compiler takes a list of optimizations to run on the code as well as the number of registers that are available on the architecture. We ran variants of the SPEC training inputs in order to keep simulation time reasonable. Our baseline timing simulator is the default `sim-outorder` configuration. A description of MIRV, our compilation methodology, and benchmark inputs is presented in the technical report of [Post00a].

All simulations were done using the SimpleScalar 3.0/PISA simulation toolset [Burg97]. We have modified the toolset (simulators, assembler, and disassembler) to support up to 256 registers. Registers 0-31 are used as defined in the MIPS System V ABI [SysV91] in order to maintain compatibility with pre-compiled libraries. Registers 32-255 are used either as additional registers for global variables or additional registers for local caller/callee save variables.

A modified version of sim-profile was used to simulate the behavior of a program compiled to use the SLAT. The simulator implements an infinite-sized SLAT with ideal replacement. Table 5.2 shows the actions that are taken at various instructions in the program. While the simulator is idealized and is not particular to an implementation, it allows us to see the potential benefits of the SLAT. Later work will address specific implementation issues.

Instruction	Action
map reg, addr	Add an entry to the SLAT. If there is a pre-existing mapping for the address in the SLAT, the data is forwarded from the previous register to the register currently being mapped. Otherwise, the data is loaded from memory.
unmap reg, addr	Remove an entry from the SLAT. If there is a previously mapped but unspilled entry, store the data from reg to the previously mapped register.
spill	If the register contains a value that was placed there by a previous map instruction, spill the value to the mapped address (home location) instead of the address specified to the stack spill location.
reload	If the previous SLAT on the SLAT stack has a mapping for this register, reload the value from its mapped address. Otherwise, reload from the specified location on the stack.
load	If any entry in the SLAT stack maps the load address, and has not been spilled, then copy from the mapped register to the load's destination register. Increment slatLoadConflicts.
store	If any entry in the SLAT stack maps the store address, and has not been spilled, then copy from the store source register to the register indicated in the SLAT entry. This implements the "callee update" register convention (a modification of "callee save". Increment slatStoreConflicts.
call	Push a new SLAT onto the SLAT stack.
return	Pop current SLAT from SLAT stack.

Table 5.2: Actions that take place at various points in the SLAT simulator.

5.6. Experimental Evaluation

This section presents our experimental results. Section 5.6.1 discusses the performance improvements possible with conventional register promotion and shows how it is limited in its applicability. Section 5.6.2 shows the performance improvement that can be obtained when values can be promoted speculatively.

5.6.1. Register Promotion

Previous work showed the performance of basic register promotion in the MIRV compiler [Post00b]. That work found that register promotion improves performance from 5% to 15% on some benchmarks. Other benchmarks perform worse with register promotion. This is due to extra register pressure caused by the promotion, which introduces spilling code.

The somewhat lackluster results for many benchmarks led us to evaluate the reasons why promotion is not performing well. The graph in Figure 5.2 shows statistics kept by the compiler which demonstrate that promotion is often limited by aliasing and side-effects. The figure shows each benchmark (along the X axis) in four different configurations. The first configuration is -O2 with separate compilation of the program's files. The compiler produces the least detailed alias information in this case. The second configuration is similar except that a simple interprocedural side-effect analysis is used to improve the precision of alias analysis at function call sites. This increases the precision of the alias analysis and allows the compiler to determine that more values are safe to promote. For these two bars, the percentages indicate the number of *static* references that fall into each category.

The third and fourth bars are similar to the first and second except that they estimate the effect of the un-promoted values by weighting each value by the number of load and store executions that would have been saved in a training run of the benchmark if the value had been promoted. Thus the percentages on the Y-axis change meaning for the third and fourth bar, because they indicate the estimated percentage of *dynamic* references that fall into each category.

The bars are divided into portions showing the reason that a promotion could not occur. The legend of the graphs are explained in Table 5.3. The last two categories–local and global side effects–are of interest in this chapter because the SLAT can aid the compiler in promoting those references to registers.

For example, for the compress95-sep bar, about 25% of static references were promoted. About 15% of values were not promotable because of local side effects, and about 60% of values were not promoted because of a global side effect. Local and global side effects are due to function call sites within the promotion region.

Overall, it is evident that of all promotion candidates, only 20% to 30% of potential promotions are actually performed. Some outliers, such as `li` have almost no promotable values, and some have a large portion of candidates that are promotable. More advanced alias analysis (shown in the second bar of each group) increases the number of promotion successes by 20% in many cases. Still, 20% to 50% of promotion candidates are not promotable using the side-effect analysis.

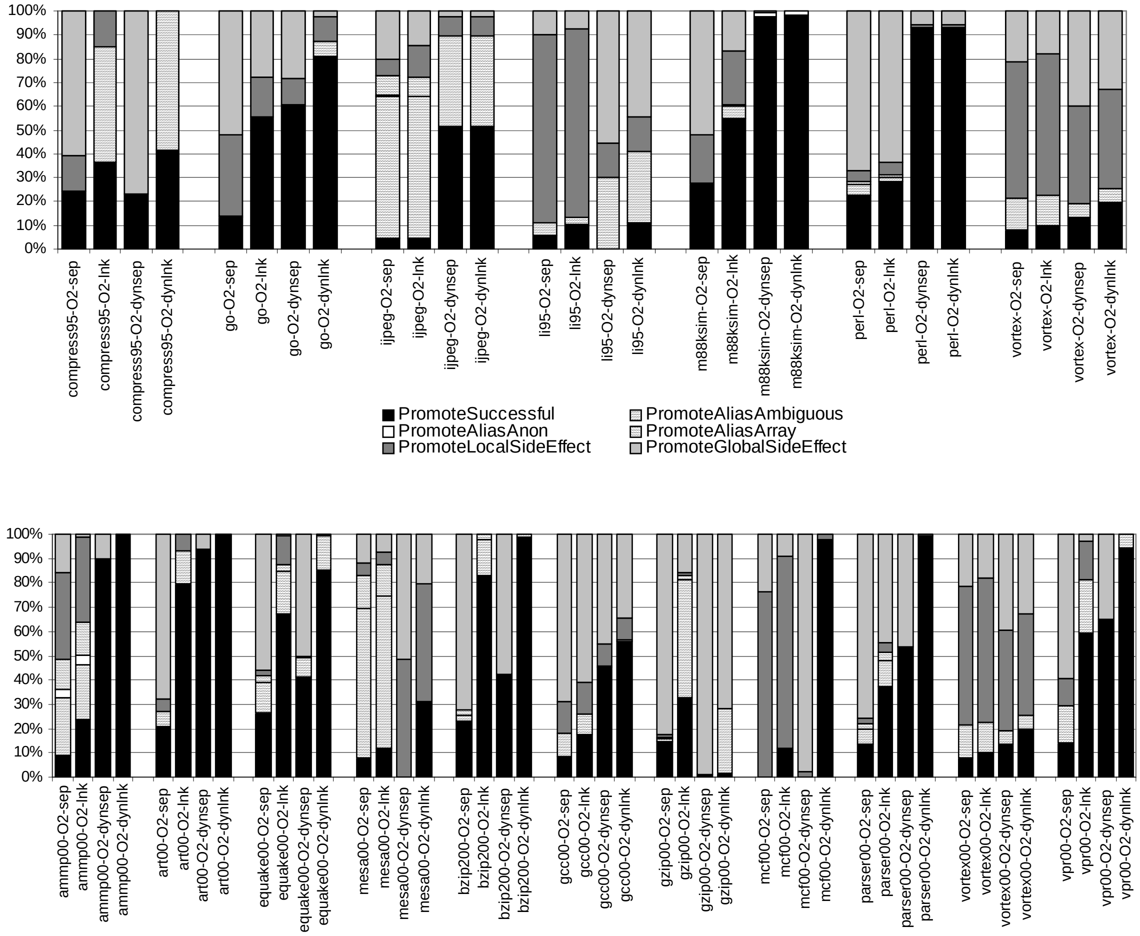


Figure 5.2: Why scalar candidates could not be promoted.
The bars shows the breakdown of reasons that promotion could not occur. All compilations use -O2 optimization. The first bar is separate compilation of modules with no interprocedural alias analysis. The second bar has interprocedural side-effect analysis information annotated at each function call site for improved alias analysis precision. This increased information in turn increases the number of candidates that are provably safe to promote. The percentages shown for these first two bars are percentages of scalar promotion candidates. The third and fourth bars are analogous except that they estimate loss in performance by weighting the counts by dynamic frequency of access to the candidate variables. The legend is explained in Table 5.3. These numbers are based on compile-time estimates and unlike later figures are only indicative of trends.

For the non-promotable candidates, the primary reason is side effects due to function calls (both local and global side effects). This implies that any mechanism that allows promotion in such regions will have to handle call sites very well.

The dynamic estimates in the third and fourth bars of the graphs in Figure 5.2 show slightly different results because the frequency count of loop bodies is taken into consideration when weighting the effect of a successful or missed promotion. The results are very benchmark dependent, sometimes showing that the static estimate was good, as is the case

Legend Entry	Explanation
Successful	The transformation was not restricted.
AliasAmbiguous	A possible manipulation of some data through a pointer prevented the transformation. In other words, there is a pointer that *might* point to something the restricts the transformation, but the compiler does not know for sure.
AliasAnon	A manipulation of code involving dynamically allocated memory was restricted by some other possible manipulation of dynamic memory.
AliasArray	A transformation involving an array was restricted by some other use of the array. Because MIRV does not track individual array elements, any reference to an array element is considered to reference the entire array.
GlobalSideEffect	A manipulation of code involving a global variable was prevented by a function call. Usually this is because a function is assumed to define and use all global variables when it is called. If, however, the compiler is performing whole-program analysis, this means that the called function references the global variable somewhere in its body, or in the body of some function further down the call chain.
LocalSideEffect	A possible manipulation of some data through a pointer passed to a function prevented the transformation. In other words, a pointer argument to the call might point to a local variable. The compiler must assume that variable is both used and defined by the call, preventing transformations across the call site.

Table 5.3: An explanation of the legend in Figure 5.2

with `vortex` and `art`. In many cases the dynamic estimate shows that the missed promotion opportunities were not significant factors in performance.

There are a significant number of promotion opportunities that are missed because of poor alias analysis. This observation led us to develop speculative register promotion, which is evaluated below.

5.6.2. Speculative Register Promotion using the SLAT

The main shortcoming of register promotion is the number of cases where promotion cannot happen because of aliasing. In this section, we evaluate the performance benefits possible from allowing more promotion via the SLAT.

Loop and Function Promotion

Table 5.4 shows the improvements possible with the SLAT. The first two numeric columns show improvement possible on top of MIRV at -O2 optimization, which includes only loop-level promotion. The numbers were collected by modifying the register promoter in MIRV to annotate the candidates that could not be promoted. Each such candidate variable reference (load or store) was annotated and each occurrence was counted during the simulation. The numbers in the table are the percentage of all load and store instructions that were thus annotated, meaning that if we had “perfect” register promotion, all of these loads and stores would have been transformed by the compiler into register references. Note that these percentages are different than shown in Figure 5.2 because those percentages are only of promotion candidates, not *all* load and store operations. One other caveat with regard to these numbers is that they are overly conservative because they count store operations that may not be necessary because the promoted variable is not actually defined in the promotion region. Therefore, several of the “improvements” in store instruction counts are actually negative, indicating that more stores were counted after the optimization than before. The actual performance will be better than these numbers show.

Even with those caveats, the `compress`, `art`, `gzip`, `parser`, and `vpr` benchmarks all exhibit significant potential for improvement for both load and store instructions–with 10% to 20% reductions possible in several cases. This substantiates our earlier conclusion that conventional promotion is unable to take advantage of many opportunities. The other results are not very significant, which is not a surprise since this optimization is very dependent on the benchmark.

The third and fourth numeric columns show what happens when loops *and* functions are considered as regions. If a variable can be promoted in a loop, it is done first. Then, if the variable is still profitably promotable over the whole function body, this transformation is made. The result is that function-promoted variables are loaded once at the top of the function and stored once before the function exits, and all other references are to a register instead of to memory. Function-level promotion increases the number of candidate loads and stores for the promoter to examine and we see a corresponding increase in the number of loads and stores that could have been eliminated with speculative promotion, but that were not removed because of aliasing problems. In this case, what has happened is that the pool of promotion candidates has been enlarged by examining the whole function body, but very few of those additional candidates are actually promoted. We verified this by comparing the overall performance of function-level promotion with the base -

Category	Benchmark	mirvcc -O2		mirvcc -O2 with function level promotion	
		Reduction in Loads %	Reduction in Stores %	Reduction in Loads %	Reduction in Stores %
SPECint95	compress	18.9	12.8	36.6	14.2
	gcc	1.3	-2.7	1.6	-5.5
	go	1.3	-1.8	1.9	-5.2
	ijpeg	0.3	-0.4	0.3	-0.4
	li	6.5	2.2	8.1	2.6
	m88ksim	0.8	0.0	3.8	-0.2
	perl	0.0	0.0	1.5	-0.1
	vortex	-1.7	-3.1	-1.1	-5.8
SPECfp2000	ammp	4.6	-0.1	4.7	-0.1
	art	13.6	12.2	13.6	12.2
	equake	4.6	-0.1	4.7	-0.1
	mesa	0.5	0.0	0.5	0.0
SPECint2000	bzip	5.3	-0.4	7.3	-1.4
	gcc	2.0	-2.5	2.3	-5.0
	gzip	24.2	12.2	31.4	18.1
	mcf	6.8	1.2	6.9	1.2
	parser	14.0	-0.5	16.9	0.5
	vortex	-1.7	-3.1	-1.1	-5.8
	vpr	7.8	-4.4	13.2	-6.3

Table 5.4: Reductions in dynamic loads and stores possible with the SLAT.
The baseline in columns 3 and 4 is compiled with loop-level register promotion The baseline in columns 5 and 6 is compiled with loop- and then with function-level promotion. The percentages give the number of loads and stores that could be removed if the promotion could take advantage of the SLAT, i.e. missed promotion opportunities are regained.

O2 configuration. There was not any significant difference (less than 1% for all benchmarks). This indicates that while function-level promotion found more candidates it wanted to promote, it could not promote most of them due to aliasing concerns. The SLAT is effective in allowing these promotions to occur.

Figure 5.3 shows what happens to promotion obstacles once speculative promotion is applied. For example, in the `compress` benchmark, the fifth bar in the figure shows that the SLAT is able to achieve just about 100% of promotions that were considered as candidates by the other experiments. It is successful at allowing promotion to happen in many more cases. Increases in promoted candidates are seen in many of the benchmarks even though separate compilation is used to produce the data for the fifth bar.

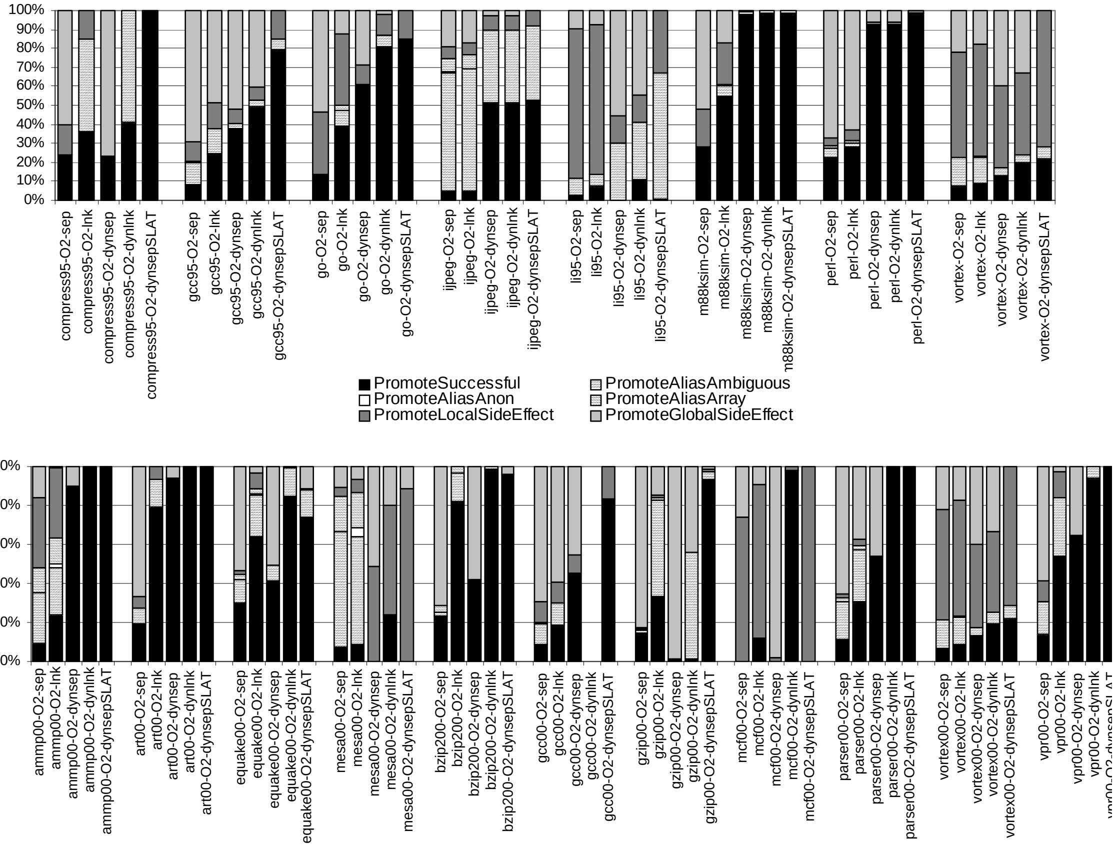


Figure 5.3: Why candidates could not be promoted with speculative promotion.
These graphs are the same as in Figure 5.2, but add one more bar to each benchmark which shows the how speculative promotion addresses the obstacles that were present in the benchmarks.

Whole-Program Global Variable Promotion

Previous work demonstrates that link-time allocation of global variables to registers is an important performance optimization [Wall96, Post00b]. The previous work has only considered “un-aliased” global variables, i.e. those whose addresses are not taken anywhere in the program. The SLAT could further improve the performance of link-time global variable allocation by allowing global variables whose addresses are taken to reside in registers for their entire lifetime. If an enregistered global variable is accessed through a pointer, the SLAT will correctly redirect the memory operation to the register file.

Experiments showed that most benchmarks are not generally improved by such a scheme. This result indicates that most global variables (or at least the important ones) do not have their address taken. This is intuitive, since the global variables are directly accessible and thus need not be used through a level of indirection. This is still promising for the SLAT, however, in a separate compilation environment. In such an environment, the compiler cannot determine which globals are aliased and which are not because modules are not visible as in our link-time, whole-program allocation scheme. Therefore, the SLAT can allow us to approach the good performance of link-time global variable allocation (as in [Post00b]) without needing to compile the whole program as a single unit.

SLAT Conflicts

We also ran experiments that quantify the frequency of conflicts in SLAT allocation and deallocation. There were five categories of conflicts that we examined:

1. The value required by a load instruction was previously mapped into the SLAT. For the most part, this happened much less than 0.5% of load instructions. There were three exceptions: `compress` (5.4%), `gzip` (3.8%), and `parser` (2.2%). This is not surprising since these benchmarks were impacted a lot by the SLAT in terms of loads and stores that were removable.
2. The destination of a store instruction was previously mapped by the SLAT. Except for `gzip` and `parser` again, this situation always occurred for less than 0.2% of stores.
3. A `map` instruction found that its address was already mapped, whether the previous mapped register was spilled or not. This occurred for 0.6% or less of `map` operations in most benchmarks. The exceptional cases were: `compress` (20%), `go` (14%), `vortex` (8%), `bzip` (7%), `gcc00` (15%), `parser` (14%), and `vpr` (11%).
4. A `map` instruction found that its address was already mapped and not spilled. `gcc95` was the only benchmark with a significant percentage of such `map` instructions (6%).
5. An `unmap` instruction found that its address was previously mapped. This is just the reverse of the previous case since the `unmap` is the counterpart of a previous `map` operation, so the percentages are essentially the same for all benchmarks.

SLAT Size Considerations

The next question we examine is how many entries the SLAT needs to achieve the performance improvements above. The simulator keeps track of the current number of SLAT entries in use and also tracks the high water mark of this number, which indicates the most SLAT entries that would ever be in use concurrently. The high water mark results are presented in Table 5.5. Except for `li` and `vortex`, none of the benchmarks require more than 34 SLAT entries to speculatively promote all aliased variables. These two benchmarks are exceptional because of their deep function call chains (`li` is a recursive descent program). Most benchmarks require less than 20 entries. This indicates that the SLAT should be effective while still very small in size. This is important since the SLAT must be fully associative. As described in the caption of the table, the third column is for loop-based register promotion, while the fourth column adds function-level promotion to the normal loop based promotion. Function-level promotion produces more candidates in the function bodies and, as we found earlier, not many of those are promotable because of alias problems. Thus the number of SLAT entries required to accommodate function level promotion is higher than for loop-level promotion–by a large margin in some benchmarks.

These numbers double-count any overlap that occurs because a variable gets allocated to the SLAT more than once. This can happen for global variables promoted in two different functions which are active at the same time on the procedure call stack. Overlap can also happen if a variable is promoted over a loop region and then the function promoter decides to promote it over the whole function body. If we corrected for this effect, the values in the graph would be even lower, meaning that an even smaller SLAT will provide the benefits we seek from speculative register promotion.

At this point it may be questioned why the SLAT would ever need more entries than there are architected registers. This is a valid question because at most only one aliased variable can be allocated to a given register at any given time, so the most active SLAT entries would be equal to the number of registers. However, at any given time, there are more values alive than there are registers because there are multiple functions “alive” on the procedure linkage stack. Each function could have promoted several values. While these values are not in the registers (they have been spilled out by the calling convention)

Category	Benchmark	SLAT Entries Actually Required	
		-O2	-O2 with function promotion
SPECint95	compress	7	9
	gcc	34	
	go	18	22
	ijpeg	23	44
	li	16	335
	m88ksim	11	15
	perl	10	11
	vortex	34	109
SPECfp2000	ammp	2	31
	art	11	14
	equake	16	20
	mesa	4	20
SPECint2000	bzip	23	21
	gcc	34	
	gzip	11	12
	mcf	5	9
	parser	26	48
	vortex	34	109
	vpr	10	17

Table 5.5: Summary of SLAT utilization.
The third and fourth columns show the maximum number of SLAT entries ever used concurrently in the benchmark, not accounting for duplicates. The third column (mirvcc -O2) is for register promotion over loop bodies. The fourth column adds promotion over whole function bodies.

they are nonetheless active in the sense that they will be coming back into registers when the procedure stack unwinds as functions are completed.

Even though we have shown that the SLAT can be very small, so far we have only shown this empirically for a selection of benchmark programs and have not proven that we can bound the usage of the SLAT to some finite number known at compile time. There are two ways to address this problem. First, the compiler can determine an upper bound on how deep the call graph will get and can it can limit speculative promotion accordingly. A second way to enforce the SLAT size constraint is to manage the SLAT with a callee/caller save convention at call sites. This could include a compiler mechanism to remap the SLAT entries after a function call which bumps them out of the SLAT (which places the management burden on the caller function) or a bookkeeping mechanism to unmap and remap any conflicting entries (which places the burden of management on the callee function.

A more detailed measure of the size requirements for the SLAT is shown in Figure 5.4. The x-axis represents the number of SLAT entries and the y-axis represents the percentage of instructions which execute under the condition of a certain SLAT size. For example, for `gcc95`, only 4 entries are required to satisfy 50% of the "execution time," as counted by the number of dynamic instructions executed. About 15 entries satisfy the benchmarks for 90% or more of their dynamic instructions.

The `ijpeg` benchmark is an exception, where 21 entries are required before the program even can begin. This points out an interesting phenomenon. In `ijpeg`, there are a number of global flags and pointers which are initialized in the `parse_args()` function. These are initialized in a section of code that loops through the command-line arguments. Thus the speculative promotion algorithm promotes these variables over the loop body. Also inside the loop body are the function calls which actually perform the primary work of the benchmark; thus the 21 SLAT entries are required for speculative promotion of the global variables at the start of the program, and these are retained in the SLAT for the entire program. References to them are redirected to and from the registers into which they were speculatively promoted. Since the (speculative) promotion happened at the beginning of the program, this is almost like link-time global variable allocation. While the later references to these globals are specified with load and store instructions instead of register specifiers, the operations themselves are redirected to the register file dynamically. In this case, a combination of hardware (SLAT) and software (speculative promotion) were used to achieve much the same effect as the link-time global variable allocation optimization, though in this case with separate program compilation. Another use of the SLAT based on this principle will be discussed in Chapter 6.

What we have found is that the SLAT optimization, though effective on some benchmarks, is somewhat more complex to implement than the link-time global-variable allocation algorithm. It also does not have quite the potential of that mechanism either. In essence, whole-program analysis has made a harder problem (speculative promotion) into an easier one (regular promotion) because of the increased visibility of the source.

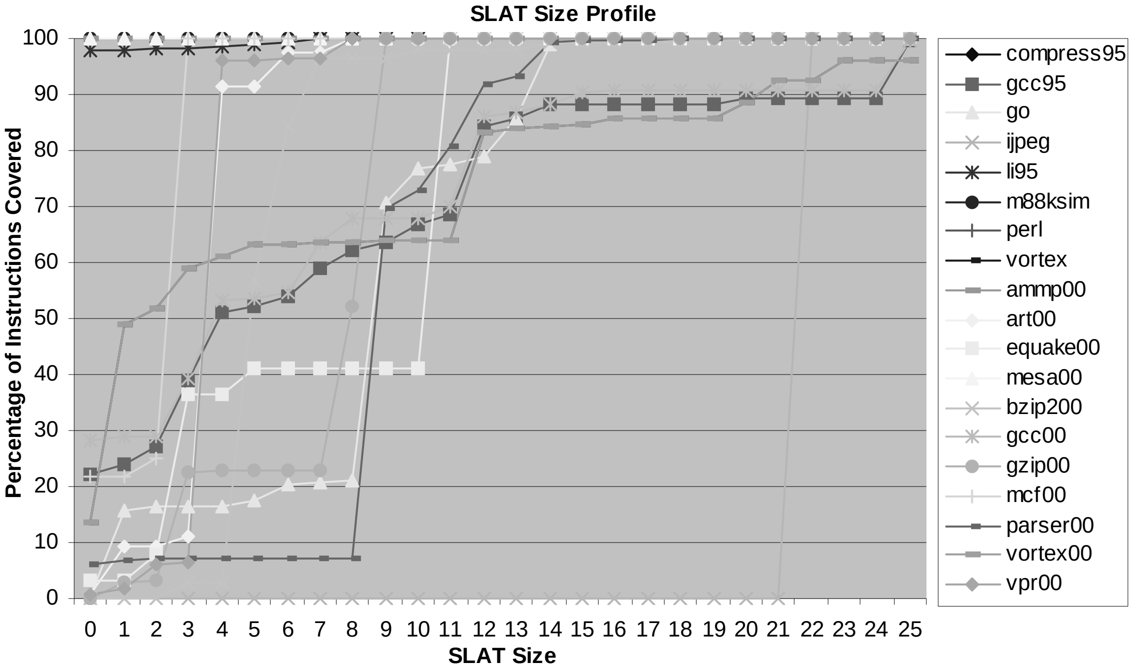


Figure 5.4: Coverage of dynamic instructions over various SLAT sizes.

5.7. Background and Related Work

This section reports on a number of proposals that combine software and hardware approaches to disambiguation, allocation, and scheduling.

Several previous proposals have discussed methods to allow register allocation for aliased variables. CRegs solves the aliasing problem by associating address tags with each register [Diet88, Nowa92, Dahl94]. These tags are checked against loads and stores to keep the registers and memory consistent. On a store, an associative lookup must update all copies of the data in the CReg array. Variable forwarding was proposed as an improvement to CRegs [Heggy90]. This technique allows the elimination of compiler alias analysis, simplifying the software side of the problem but complicating the hardware because a value can be mapped to any registers in the register file. Chiueh proposed an improvement on both CRegs and variable forwarding [Chiu91]. Aliased data items are kept in the memory hierarchy (data cache) and accessed indirectly through registers. The registers contain the address of the value and the compiler specifies a bit on each operand in the instruction to direct the hardware to use that register indirectly. This technique was independently discovered by the present author [Post99].

The weakness of CRegs is that writes must associatively update several registers. The SLAT does not require this associative write-update to the register file because the compiler guarantees that only one copy of the data is mapped to a register within a function. This vastly simplifies register access compared to CRegs.

Nicolau proposed a purely software disambiguation technique where a load could be scheduled ahead of potentially dependent stores [Nico89]. This technique is called runtime disambiguation because the hardware checks conditions at runtime to determine if a conflict has occurred.

The Memory Conflict Buffer (MCB) is designed as an extension of Nicolau's runtime disambiguation. It allows the compiler to avoid emitting explicit (software) checks of address operands [Gall94, Kiyo]. Instead, addresses that need to be protected are communicated to the hardware by special load operations and then special check operations ask the hardware whether a conflict has occurred for the given address. Hardware does the address comparisons instead of software. Like for runtime disambiguation, the goal is to perform code scheduling in the presence of ambiguous memory operations.

The SLAT is different from the MCB in that it must retain information across function calls to be effective–as was shown, this is important because many aliases are due to assumed side effects of function calls, so that SLAT must handle function calls elegantly. The information stored in the MCB is not valid across function calls [Gall94].

The IA64 architecture provides hardware support for compiler-directed data speculation through use of an Advanced Load Address Table (ALAT) [IA6499]. It allows static scheduling of loads and dependent instructions above potentially aliased stores. The compiler is responsible for emitting check and fixup code for the (hopefully rare) event that a conflict occurs.

The SLAT is different than the ALAT in a number of respects. The most notable difference is that the hardware must compare not only store addresses to all SLAT entries (as with the ALAT), but in addition it must compare all load addresses as well. This is because the most current value for the memory location could be housed in a register and any loads that access that memory location need to receive the current value. The ALAT cannot provide this functionality because the hardware only checks the addresses of store instructions with the entries in the ALAT.

Another difference is that the SLAT must retain all the information ever entered into it whereas ALAT entries can be replaced because of overflow, conflicts, or context switches. This is because the ALAT requires an explicit check instruction to determine if the fixup code needs to be run. If an entry is missing from the ALAT, the check instruction runs the fixup code. Thus the ALAT is “safe” even when it loses information. On the other hand, if the SLAT “loses” an entry, load and store instructions could be executed without detection of conflicts, which would produce incorrect program output.

Another difference between the SLAT and ALAT is that SLAT fixup is not initiated at the point of transformation but at the point where the conflict occurs. For the ALAT, fixup is always initiated at the point of the original load (which has been converted to a check load). For the SLAT, since the correct data is in a register, the hardware can forward the data for a load from the register or for a store to the register.

Transmeta Corporation recently introduced a line of processors that is designed to run unmodified x86 programs using dynamic binary translation [Klai00, Wing]. Capability similar to the ALAT is provided by special hardware and instructions to allow load and store reordering. Two instructions are necessary for this: `load-and-protect (ldp)` and `store-under-alias-mask (stam)`. The `ldp` instruction “protects” a memory region. The `stam` instruction then checks if it would store to a previously protected region. If it would, it traps so that fixup can be performed. The main purpose of this system is to allow Transmeta’s code morphing software to allocate stack variables to host registers.

The SLAT differs from this approach in that it is designed for a static compilation environment, hardware corrects conflicts instead of taking an exception, and memory does not necessarily need to be kept up to date since the latest value is in the register.

5.8. Conclusions

This chapter has described the design of the store-load address table, its use in a new optimization we call speculative register promotion, and the reductions in load and store operations possible when using this optimization. We began by showing that register promotion was often limited by compiler alias analysis. The number of loads and stores can be significantly reduced for several of the benchmarks with the addition of a SLAT and speculative register promotion–up to 35% reduction in loads and 15% reduction in stores. Applying the SLAT to link-time global variable allocation does not produce much benefit for most benchmarks. It is more important in this case to note that the SLAT effectively allows link-time allocation even in the face of separate compilation, so that the SLAT can achieve most or all of the benefit of link-time allocation while doing so in a separate compilation environment. Finally, we showed that the SLAT can be modestly sized and achieve the benefits reported here.

There are several important avenues of future work. Other uses of the SLAT should be examined. Future work should also investigate other ways that the hardware can help the compiler do aggressive, potentially unsafe operations.

Chapter 6
Effect of Compiler Optimizations on Hardware Requirements

6.1. Introduction

The techniques from Chapters 3, 4, and 5 have promised significant speedups in the execution of the codes examined in this dissertation. The present chapter continues these studies by examining the trade-off between hardware and software implementations of the proposals of the previous chapters. The chapter studies some of the effects of the optimizations we proposed in Chapter 3 on cache performance. It also examines the optimizations in light of cache performance and issue width to determine how much additional microprocessor hardware it takes to achieve performance equivalent to that produced by the optimizations. We do this using two metrics, the issue-width-equivalent (IWE) and the cache-latency-equivalent (CLE). These metrics emphasize the relative value of adding processor hardware versus adding compiler optimizations. In general these metrics intend to address the issue of compiler complexity versus hardware complexity, but this work will not fully examine that trade-off.

In addition, this chapter examines some characteristics of memory operations, particularly comparing them before and after the optimizations presented in Chapter 3. What are characteristics of most common accesses and misses? How often are memory locations reused? How do accesses and misses correlate with data address and load/store program counter? What is left to be done after these optimizations? The answers to these questions serve as a basis for suggesting other optimizations and extensions to the work in the previous chapters.

The result of this study is a rough characterization of memory operations into several classes:

1. Those which can be transformed to register operations. These were considered in the previous chapters.
2. Those which are highly reused and are thus good candidates for further penalization.
3. Those which are “problematic.” Such operations are classified this way because they come from often-used source code which has complicated memory address calculations. Examples of such code, which either accesses the memory frequently or misses the cache frequently, are shown at the end of the chapter.

6.2. Effect of Compiler Optimization on Cache Performance

This chapter first looks at the effect of aggressive optimizations on data cache performance. The L1 data cache miss rate is typically quite low for the SPEC applications used in this study, as shown in Figure 6.1. This is to be expected with the small data sets that were used in these experiments. However, an interesting trend is observed in the miss rate. The aggressively optimized binaries (best112) always have a worse miss rate than the O2-baseline binaries. While this is misleading, because the optimized binaries perform better than the baseline ones, it is expected for two reasons. First, the optimizations remove memory accesses, thus reducing the denominator of the misses/accesses equation. Second, the accesses that are removed are primarily cache hits (see Chapter 3). Therefore, the numerator of the cache miss rate equation stays the same. The number of misses has not increased, but the miss rate has increased.

Another metric commonly used to characterize cache misses is misses-per-thousand-instructions (MPI). This is shown in Figure 6.2. Except for `art`, none of the benchmarks have more than 20 MPI. Again, the heavily optimized binaries show a slight increase in MPI. This is because the path-length of the program, i.e. the number of dynamic instructions, has been reduced.

For an 8KB data cache, the graphs are similarly shaped, but shifted up by 2-5% miss rate and 5-10 MPI.

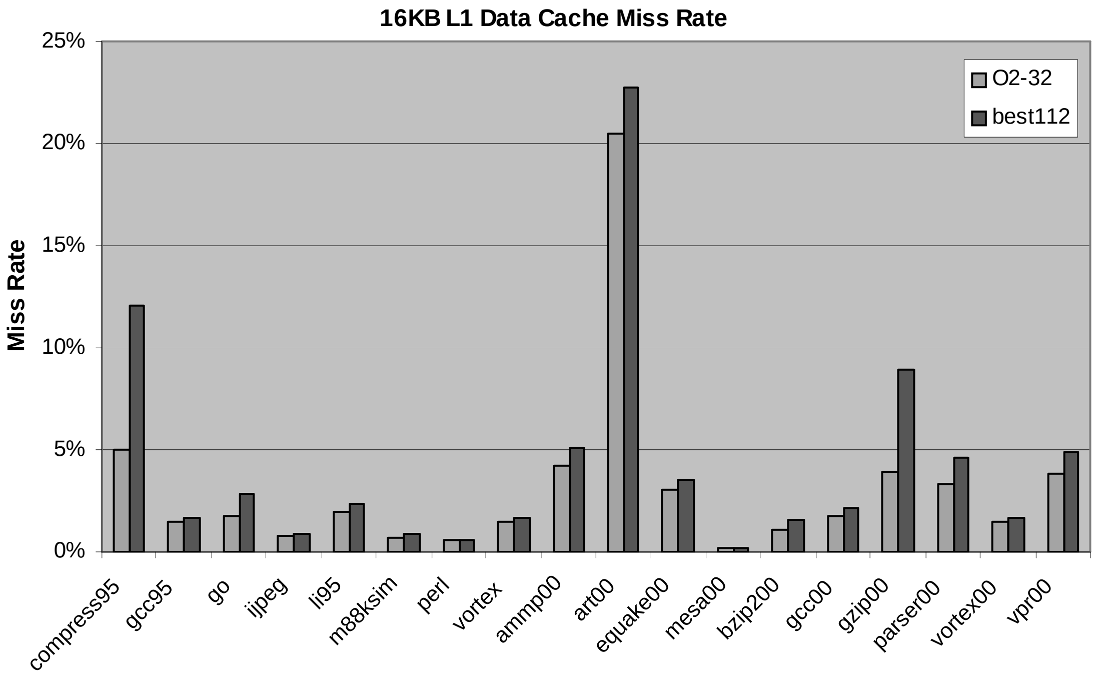


Figure 6.1: L1 data cache miss rates before and after optimization.

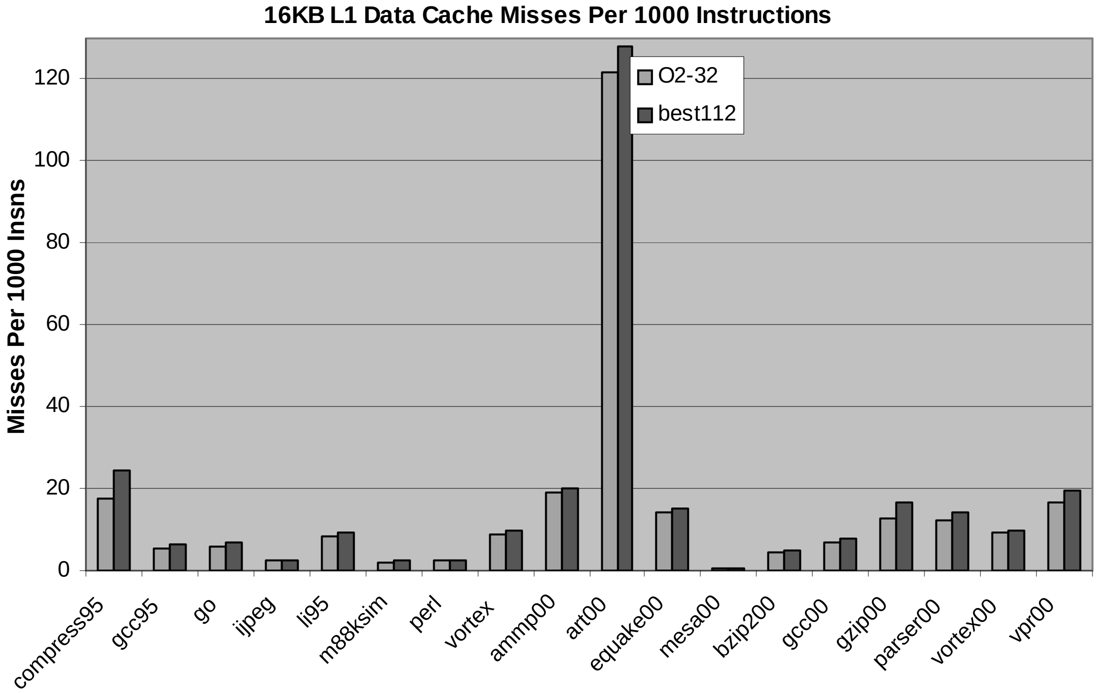


Figure 6.2: Misses per 1000 instructions before and after optimization.

Processor/ Microarchitecture	L1 Data Cache Size	Through-put	Latency
P6 [Gwen95]	8K	1	3
SB-1 [Gwen98]	32K	1	1
21164 [Gwen94a]	8K	1	1
21264 [Gwen96]	64K	1	2
Transmeta 5400 [Half00b]	64K	1	2
PIII [MDR99b, Carm00]	16K	1	3
P4 [Carm00]	8K	1	2
PA8000 [Gwen94b]	1MB	1	2
K7 [Deif98]	64K	1	3

Table 6.1: Cache configurations of some modern processors.

6.3. Hardware-Compiler Trade-offs for the Cache

The cache configuration of modern microprocessors varies widely as designers have made different trade-offs. A chart of several cache hierarchy configurations found in modern processors is shown in Table 6.1. Many processors have caches whose access latency is 2 or 3 cycles. All the simulations performed in this dissertation so far have assumed a single-cycle of cache latency. This puts our optimizations in the least favorable light because the cache is very fast, so our optimizations have less opportunity to improve the performance. In this section, we show what happens to the optimizations of Chapter 3 when the cache latency is increased to 2 or more cycles.

Figure 6.3 shows the performance of the best112-optimized benchmarks as L1 data cache latency is varied (all designs modeled are fully pipelined, so the throughput is 1). The baseline, to which each benchmark is normalized, is always at a performance level of 1. The first bar, b112-8-1, is for the best112-optimized binaries with an 8KB L1 data cache and 1 cycle access time. Ignoring the effect of the larger register file on cycle time, this configuration shows a performance improvement of slightly negative to over 30% depending on the benchmark.

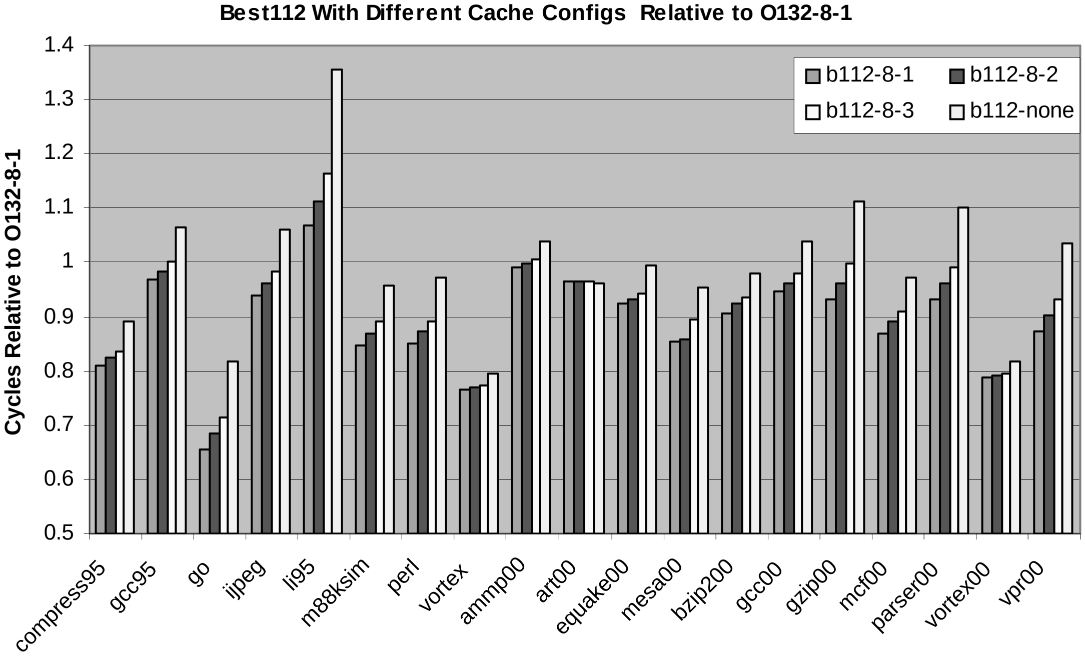


Figure 6.3: The effect of cache latency on performance.

As the cache latency is increased to 2 and then to 3 cycles, this performance benefit is reduced, but for most of those cases a 3-cycle L1 latency still does not overcome the benefits due to the optimizations. In fact, we also tried the b112-none configuration, which has no L1 data cache. The unified L2 cache is the only data cache in this model and has a 6-cycle hit time. Many of those configurations with optimized binaries still perform better than the baseline. This shows that the cache-latency-equivalent (CLE) for many of the benchmarks is more than 6 cycles. That is, our optimizations can be used instead of a fast L1 data cache and still achieve comparable performance.

This result suggests a trade-off between compiler optimization and hardware. More optimization can be used to overcome slow cache. In the extreme, the data shows that no L1 data cache is necessary at all.

These results are somewhat surprising, but there are a couple of reasons for them. First, many loads are latency tolerant and as long as they are satisfied within the second-level cache hit time, performance will not be degraded much [Srin98]. Second, other non-ideal effects in the processor attenuate the effects of cache latency. For example, imperfect branch prediction effectively shrinks the size of the instruction window because of frequent processor restarts. This reduces the pressure on the cache to provide results immedi-

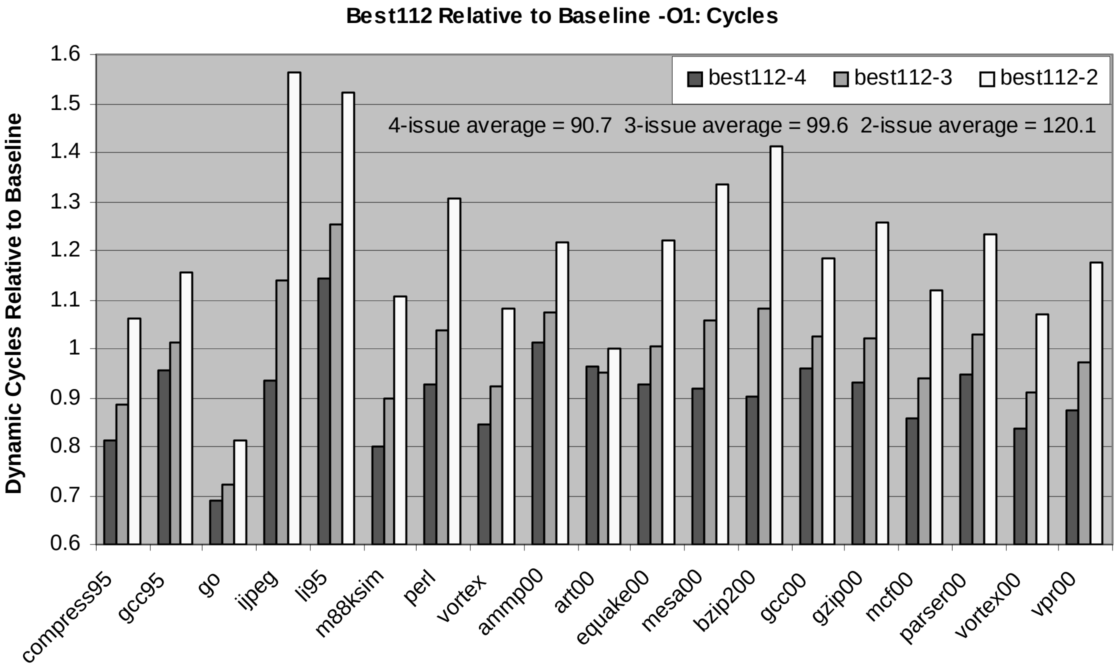


Figure 6.4: The effect of issue width on performance.

ately. These results are also biased towards the SPEC application suite that we have considered in this work, primarily because there are not major problems with cache misses in these benchmarks.

6.4. Hardware-Compiler Trade-offs for Issue Width

Figure 6.4 shows the performance of the best-optimized benchmarks as issue width varies from 2 to 4 instructions per cycle. The 4-issue processor averages a 10% improvement when running a heavily optimized (best112) binary. Ignoring any increased cycle time due to the larger register file, this is the improvement due to compiler optimization. If the processor is restricted to issue 3 instructions per cycle, then it averages about 2% improvement over the original 4-issue baseline. When issue is restricted to 2 per cycle, performance averages 18% worse than the 4-issue baseline, though several benchmarks still perform better than the baseline.

This shows the trade-off between compiler optimization and hardware complexity, similar to the last section. According to the above data, the optimizations are worth between 1 and 2 issue widths. We call this the Issue Width Equivalent (IWE) metric for the optimizations. For go, then, the IWE is higher than 2 because it still performs better with optimizations on a 2-issue configuration than without optimizations on a 4-issue configuration. For bzip, taking away one issue width causes performance worse than the baseline. So the IWE of the optimizations for this benchmark is less than 1.

It is instructive to note that most benchmarks suffer greatly when reduced to a 2-issue processor, even with the aggressive best112 optimizations. There is simply not enough execution bandwidth to satisfy the needs of the program.

More formally, the IWE metric can be defined as follows. It is based on the following equation:

executionCycles(base, 4-issue) = executionCycles(optimization, n-issue).

The baseline binary and machine configuration is represented on the left-hand side of the equation, the optimized binary and smaller machine configuration on the right-hand side. The variable n is the issue width of this smaller configuration. To determine the IWE, n is decreased until the equation above becomes true, i.e. that the performance of the optimized binary becomes the same as the performance for the original configuration. At this stage, the following equation determines the issue-width equivalent: IWE = trunc(4 - n). For m88ksim, n is almost 2. So, $\text{IWE}_{\text{best112}}$ = trunc(4-2) = 2. The optimizations are worth about 2-issue widths (2 pipelines).

The exact IWE would be fractional, but it does not make sense in reality to make a 3.5 issue processor. So we simply cap the IWE at the largest integer less than 4-n.

6.5. Characterization of Memory Operations

This section reports on a number of characteristics of the memory operations both before and after optimizations. The goal is to gain some understanding of program behavior with respect to memory operations and to suggest possible optimizations that can take advantage of the observed characteristics.

6.5.1. Experimental Setup

The experiments in this section were performed on a modified version of the SimpleScalar sim-cache simulator. A number of new statistics were added to the simulator to study different characteristics of the memory operations. For every address touched, the simulator keeps a record of how many stores, loads, misses were to that address. It also keeps track of access size to see if addresses are used by more that one size of load or store operation. It produces distributions of the top 30 addresses by sorted by frequency of access and by frequency of L1 data cache misses. It also prints out what addresses are only accessed once, the number of unique addresses touched, the average address reuse, the locations that are never stored or loaded, and the amount of data touched by the program.

6.5.2. From the Viewpoint of the Memory System

The first measure that we examine in this section is the memory footprint of the benchmarks that we are using in this dissertation. The raw numbers, in kilobytes, are shown in Table 6.2. They were computed by keeping track of every 32-bit word in the data memory space. Writes to bytes within that word are simply counted as accesses to the word. The footprint is simply the number of words touched by the program multiplied by 4 bytes per word. The benchmark data sets vary on the small end from 220KB for `li` and 345KB for `go` up to over 20MB for `bzip200`. An earlier section showed how the 16KB cache handled these data sets–very well considering the large size of some of them. Evidently the working set at any one time is quite small for most of these benchmarks. Go is interesting in that it is often considered one of the more realistic of the SPEC benchmarks. It has a very small memory footprint, which would indicate that it does not stress the memory system very much. This is not always the case, however, because in spite of a small data footprint, the `art` benchmark has a very high proportion of L1 data cache misses.

Figure 6.5 shows the percentage of memory accesses that are attributable to the top 30 heavily accessed addresses. For example, for the compress benchmark compiled with O2 for a 32-register machine (O2-32), the 30 most frequently touched memory addresses account for about 75% of the memory accesses. This simply means that if those 30 addresses were kept in a cache, 75% of all memory accesses would be serviced from that cache. This phenomenon is called *address reuse*. Our experiments are counting address reuse for every load and store operation without regard for where the data is located in the

Benchmark	Footprint (KB)
compress95	597
gcc95	678
go	345
ijpeg	1,123
li95	220
m88ksim	5,173
perl	1,138
vortex	8,184
ammp00	222
art00	1,334
equake00	6,117
mesa00	14,875
bzip200	20,338
gcc00	1,177
gzip00	2,520
parser00	7,726
vortex00	8,133
vpr00	1,459

Table 6.2: The memory footprint of selected SPEC benchmarks.

memory hierarchy at the time it is loaded or stored–whether in a load/store queue, L1 data cache, L2 data cache, or main memory.

For some benchmarks (`compress`, `m88ksim`, `mesa`, and `gzip` in particular), 40% or more of the memory accesses occur to 30 or fewer addresses. For most of the other benchmarks, less than 20% are from those addresses. These 30 addresses are responsible for almost none of the misses in the programs, which is not surprising since they are heavily used.

This data points to the possibility of using a specialized cache for these heavily-used addresses. The addresses could be allocated to registers and protected by the SLAT, where this structure is used differently than proposed in Chapter 5. We did some simple experiments along this line but found it difficult to determine, at runtime, which addresses to put into the special cache structure or SLAT. With profiling, some of the addresses turned out to be in data structures whose addresses are available at compile time. Many of the addresses, however, are from dynamic memory allocation and their particular values

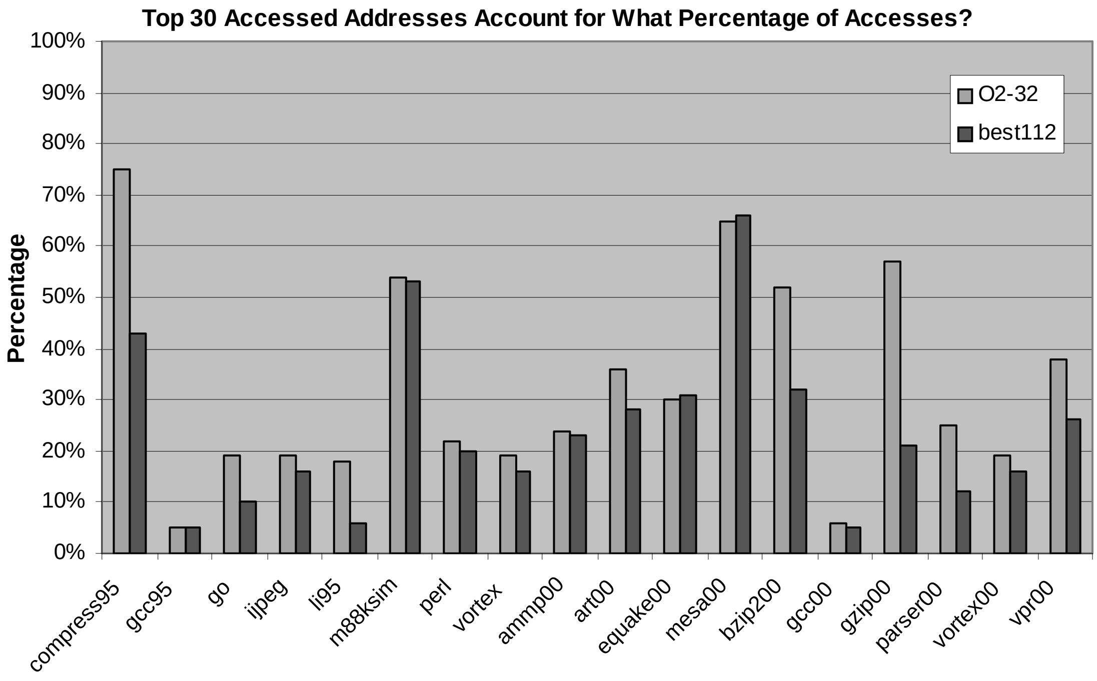


Figure 6.5: Memory accesses due to the top 30 frequently used addresses.

cannot be determined at compile time. This would be similar to placing heavily missing cache lines into a special cache; here we are suggesting the possibility of putting heavily used addresses into a cache, without regard for their temporal or spatial locality.

The other trend is interesting to note is that heavy optimization (best112) almost always causes the address reuse to decrease. For `compress`, this is a significant effect because a number of key global variables are allocated to registers and all the accesses to them are eliminated from the memory hierarchy. The memory operations which remain are primarily to the input and output buffers that are very heavily used in `compress`.

For the other benchmarks, the effect is not so significant, yet the trend indicates that the optimizations are removing heavily reused addresses from clogging the memory hierarchy and effectively caching them in the register file. Another way to see this is that the optimizations are allocating some data to registers such that the remainder of the accesses are, on average, more spread out over the address working set.

Address reuse can be shown another way, as in Figure 6.6. The graph shows 4 bars, two for O2-32 and two for best112. All the values for address reuse are computed based on 32-bit words; the value of each bar is computed by the number of data accesses made divided by the number of unique words touched during the program execution (the footprint). The result is staggering in a number of cases–for go, li, art, gzip, parser, and vpr, the reuse averages 600 to 1800 uses per address over the benchmark run. Such high reuse is why caching works so well.

The first bar shows the reuse for an O2-32 binary and the second when optimizations are applied. The next two bars are similar but eliminate the addresses which are only used one time. This removes some skew to the average which is present, particularly in those benchmarks which have a high percentage of addresses that are only touched one. For example, the go benchmark exhibits over twice the average reuse when such addresses are removed from the metric. The reason for this will be examined later.

From the graph, it is apparent that the optimizations always reduce address reuse, sometimes quite significantly, by moving heavily reused items from the memory hierarchy to the register file (where we do not count reuse). This reduction in address reuse indicates a general trend in computational redundancy: compiler optimizations tend to reduce program redundancy. This has implications for address and value prediction because as more aggressive optimizations are used and discovered, program redundancy is reduced and therefore those techniques will have diminishing applicability in the future.

This data also indicates an interesting trend that memory addresses fall into one of two rough categories–either the address is heavily reused or it is used once at that is it.

One more metric we examined with regard to data addresses is the percentage of misses due to the top 30 missed addresses. This data is shown in Figure 6.7. As far as data addresses are concerned, the top 30 missing addresses are not generally responsible for more than 5% of the misses in the program, though for vortex these addresses are responsible for 25% of the misses. This indicates a possibility for optimization in vortex by caching the most missed data addresses in a special cache (the “miss cache”), or locking them in the primary data cache, so that they do not miss anymore. Unfortunately, these addresses generally account for 1% or less of the total memory accesses in the program. The parser benchmark is the exception, where about 4% of the accesses are represented in the top 30 missed addresses. Optimization does not significantly change the situation.

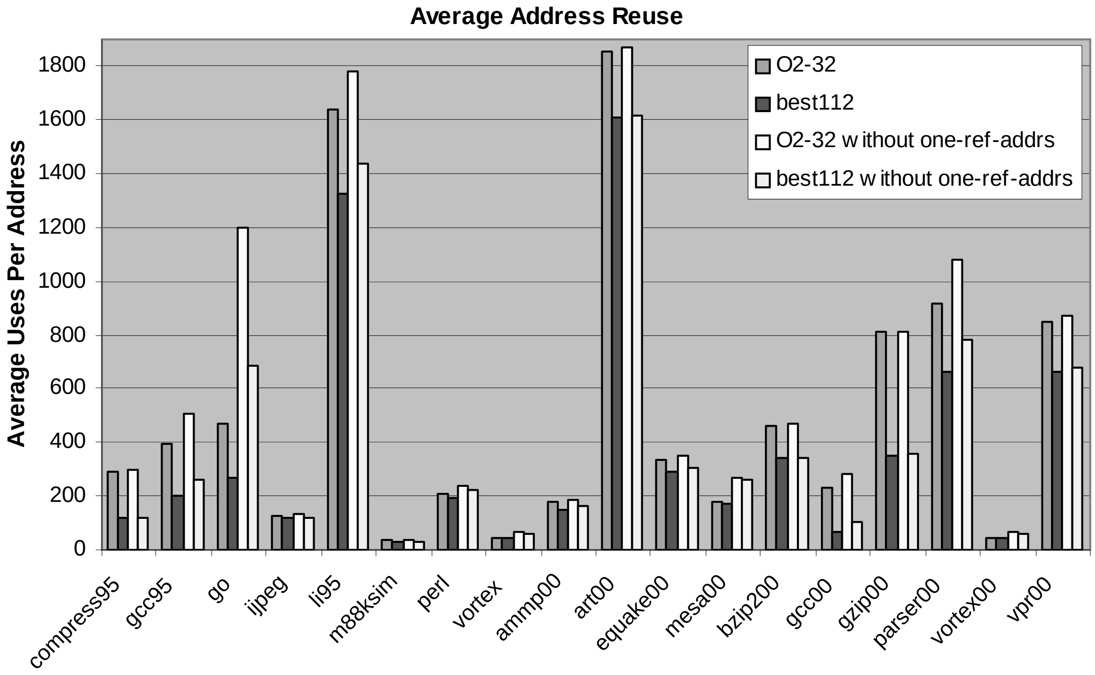


Figure 6.6: Address reuse.

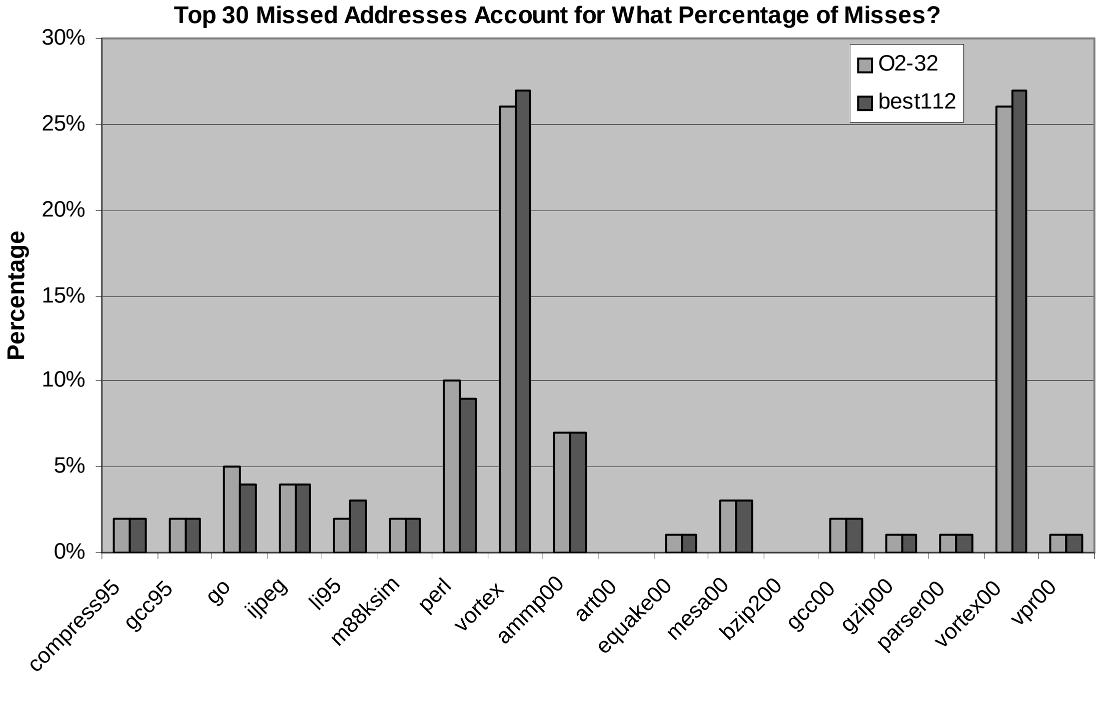


Figure 6.7: Misses due to the top 30 frequently missed addresses.

Figure 6.8 shows another property of the memory access in the SPEC benchmarks. For several of the benchmarks (`go`, `vortex`, and `mesa` in particular), a significant per-

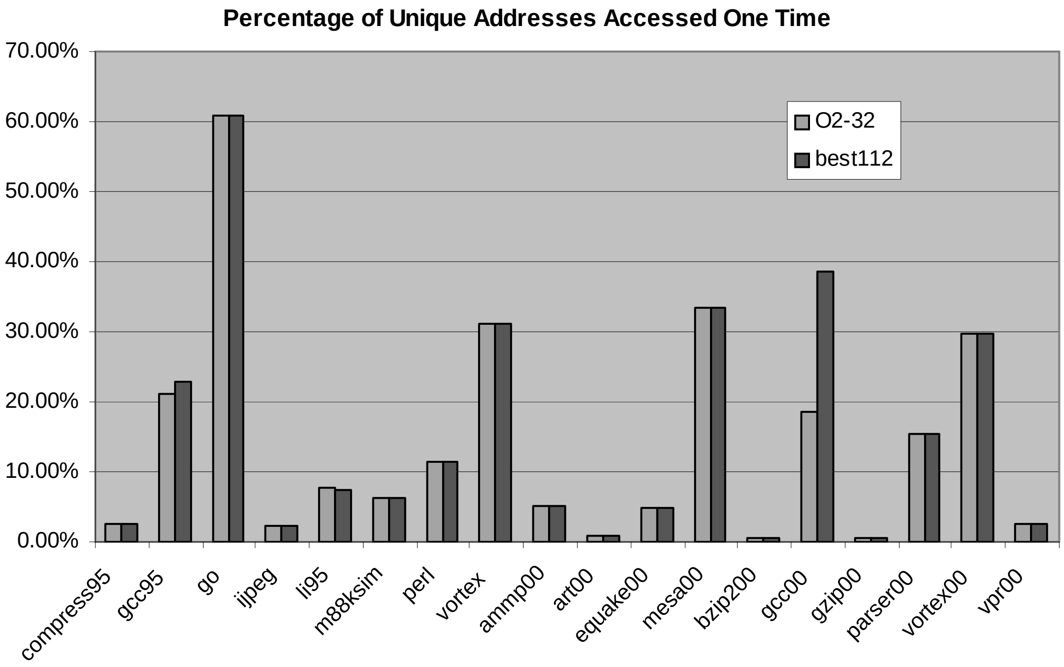


Figure 6.8: Fraction of address space referenced only one time.

centage of memory locations are only touched one time. This was a strange phenomenon which we spent some time examining to determine its cause.

For the `go` benchmark, the cause was a number of over-initialized arrays. The arrays were extended beyond what was ever used in our benchmark run; the unused values were initialized once at the start of the program. Thus, 60% of the data footprint of `go` is only touched once, and that with store instructions at the start of the program. Relating this to the data footprint information in Table 6.2, we see that only 40% * 345KB, or 138KB of data is actually important to the program’s execution. This small working set explains why the cache performance of `go` is so good.

For `mesa`, a similar problem exists with a number of `memset()` calls which set memory which is never then used. The image used in our test run of `mesa` contained only blue pixels (no red or green) and so the program’s behavior is a little skewed. `Vortex` is similar.

The remainder of the benchmarks usually do not exhibit this phenomenon for more than 10% of their address space.

Special care was necessary in the simulator to count the number of such once-referenced addresses in the presence of accesses to bytes, half-words, words, and double-words. For example, if a memory region is initialized with `memset()`, which treats memory as an array of integers, and then the region were read as an array of chars, 3 out of the 4 addresses (the last three bytes of the word) are initialized under the guise of a larger data type. Our simulator examines addresses exclusively at the 32-bit word granularity, and treats byte reads and writes and other odd-size accesses accordingly.

Figure 6.9 shows the fraction of the data memory footprint which was never stored or never loaded from. The fraction of never-stored locations is generally very small. These are due to static arrays, for example, that are initialized at program load time and never stored again. However, the percentage of locations that are never loaded is quite high. This indicates either memory locations that were initialized and never used (like in the `go` benchmark as described above), or program results which are generated but never again examined by the program. This is the case for `vortex` which sets up a database and then queries portions of it. The effect is doubled for `m88ksim`, which loads a binary and executes a portion of it, and the binary itself writes some locations but never reads them. The `compress` benchmark compresses a text buffer A into buffer B, un-compresses B into C, and then compares C to A. However, it only compares the first and last characters of C to A, so it does not load any values from the middle of C. Such addresses should not be kept in the cache, suggesting that a special store instruction could be used which bypasses the L1 data cache and stores to a more distant level in the memory hierarchy. Of course, this behavior is mainly due to the artificial “benchmark-nature” of this program.

Combining this data with that in Figure 6.8, we see that even though a large fraction of memory in some benchmarks, such as `ammp`, are never loaded, only about 5% of the memory space is touched a single time. This means that many locations are never loaded but are stored multiple times. Otherwise many locations were never loaded and only stored once, there would be more than 5% of the address space touched only once.

Another problem that we discovered was that the SimpleScalar simulator, when it encounters a `read()` system call, executes the system call in a single simulation step, without actually running any code for it. So, `read()` reads into memory a large chunk of

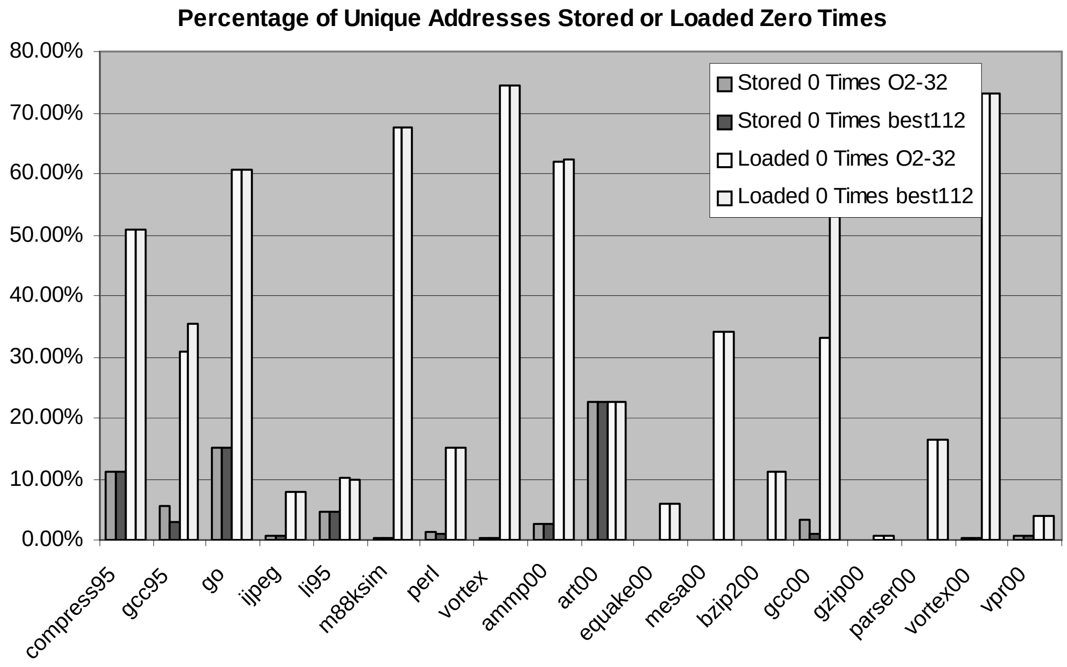


Figure 6.9: Fraction of address space never loaded or never stored.

data without using any store instructions to touch the modified addresses. From our simulator's point of view, later loads from those locations will look as if they are coming from un-initialized (never stored memory).

6.5.3. From the Viewpoint of the Instruction Stream

Figure 6.10 shows the percentage of accesses that come from the top 30 heavily used load and store instructions. Typically, at least 20% of accesses are concentrated in this small number of load and store instructions. For `compress`, `m88ksim`, `art`, `gzip`, and `vpr`, these numbers vary from 50% to 70%. When heavy optimizations are turned on, the most frequently used memory instructions account for an even higher percentage of memory accesses. Those variables which have been allocated to registers by the optimizations are primarily scalars which tend to be accessed in a wider variety of locations in the program than array elements, for example, which tend to be accessed by fewer memory instructions which are executed heavily in loops. The result is that, with optimization, the memory accesses become more clustered on a smaller set of program load and store instructions.

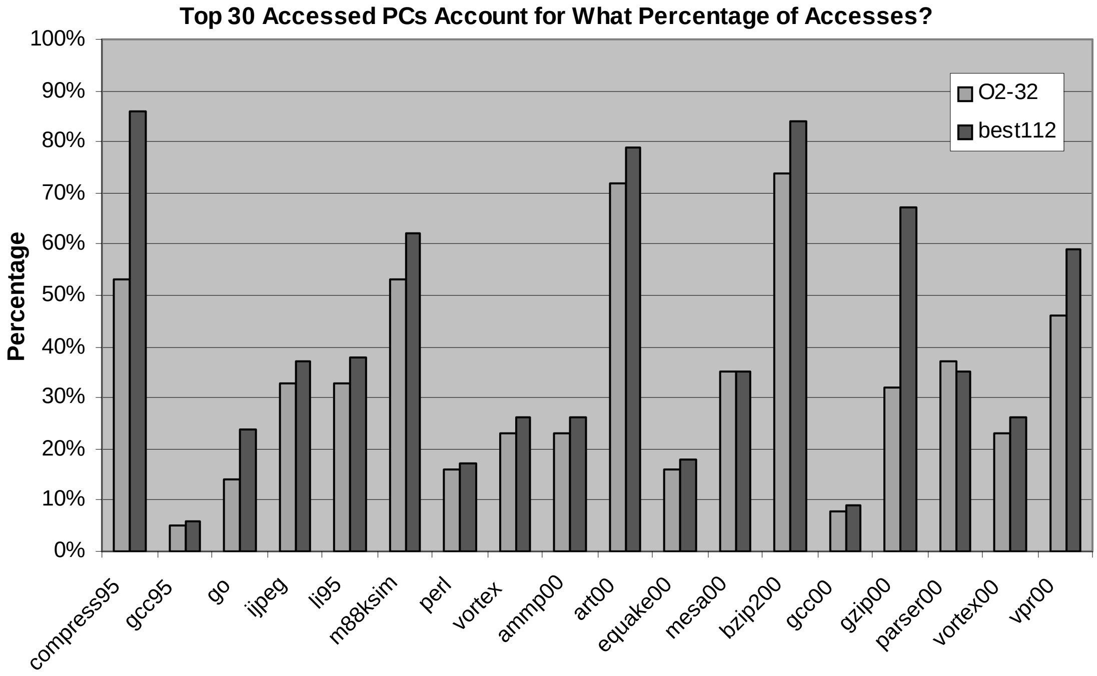


Figure 6.10: Memory accesses due to the top 30 frequently executed PCs.

Unlike the heavily used data addresses, which tend not ever to miss in the cache, the heavily accessed PCs tend to account for quite a high percentage of cache misses, often 40% or more. This data is shown in Figure 6.11. The load and store operations responsible for these accesses should be the primary targets for optimizations such as register allocation, register pre-loading, etc.

Figure 6.12 shows the correlation between program counter and misses. The top 30 program counters which incur L1 data cache misses are shown in the figure; the metric is the total percentage of misses for which they are responsible. This is the most highly correlated data that we have seen thus far–the top 30 missing program counters are often responsible for 70% or more of the misses. Optimization has little effect on this, because the optimizations we apply generally move cached items into the register file.

The primary reason for this highly correlated behavior is that load and store operations to arrays and linked structures such as trees, which occur in loops, are responsible for the majority of cache misses. Thus a small number of load and store instructions is found to be responsible for a large number of the misses.

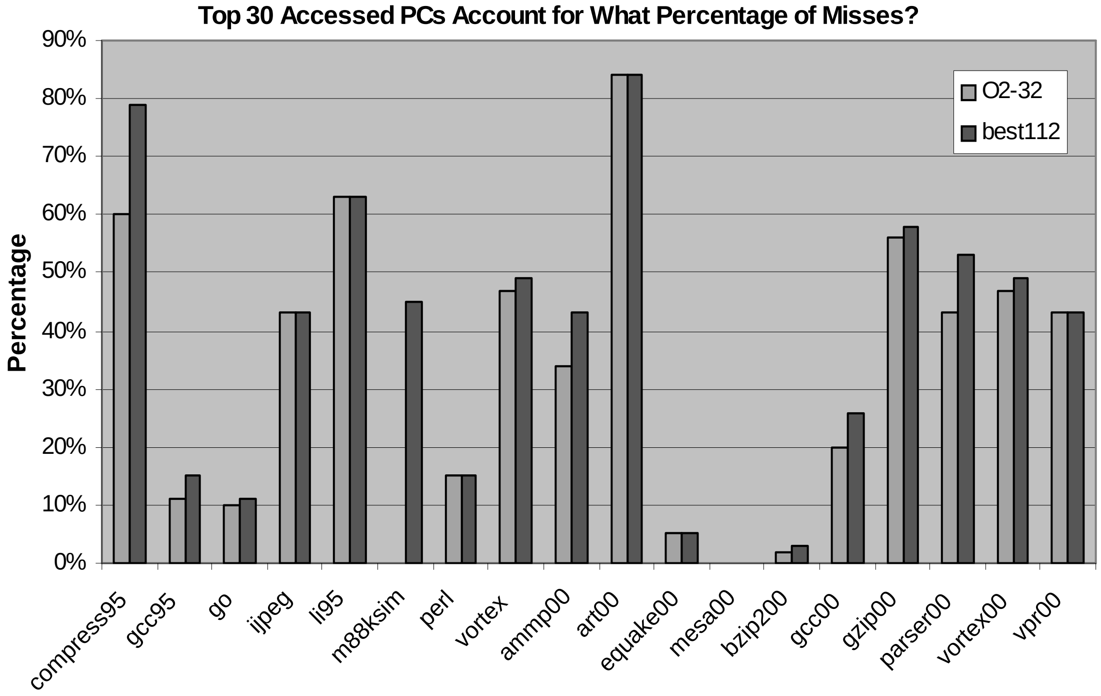


Figure 6.11: Misses due to the most frequently used memory instructions.

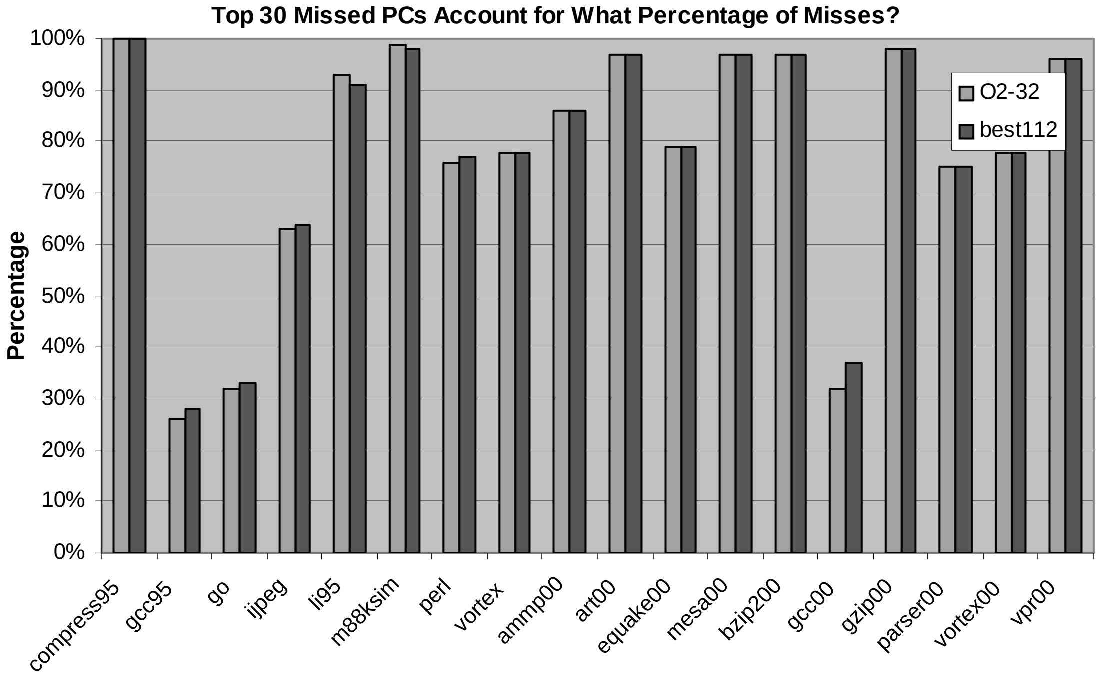


Figure 6.12: Misses due to the top 30 frequently missing memory instructions.

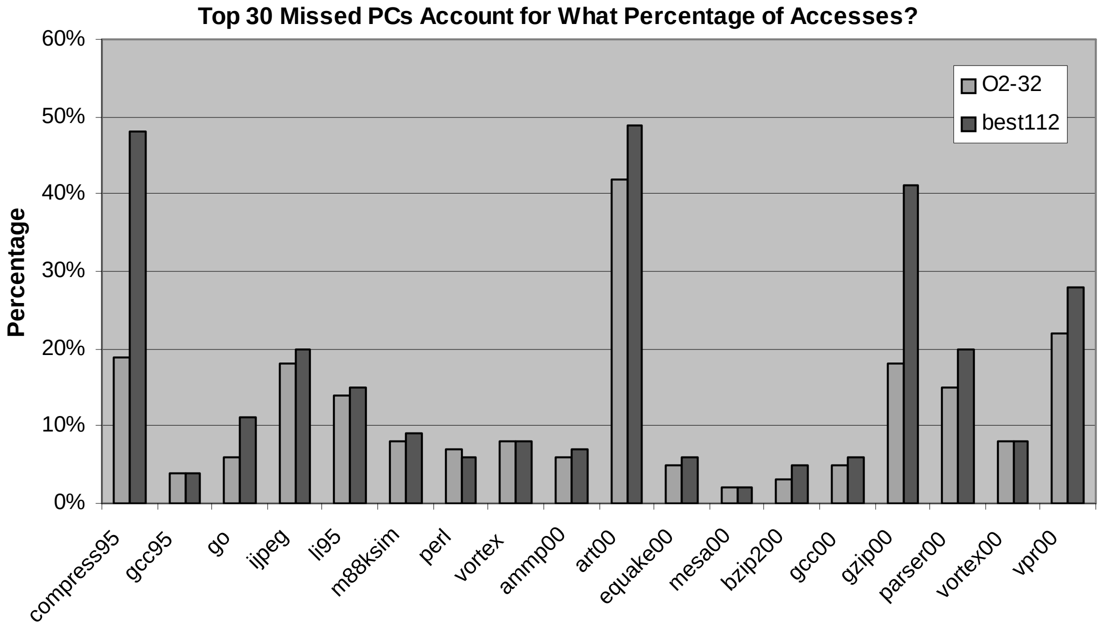


Figure 6.13: Accesses due to the most frequently missed program counters.

The top 30 program counters responsible for misses are also responsible for a sizeable fraction of total cache accesses, as shown in Figure 6.13, sometimes as much as 40%.

This data clearly suggests the following statement is true: if a load operation has missed, it will likely miss again. If has missed a lot in the past, it will likely miss a lot in the future. This shows the importance of finding the program locations of the load and store instructions responsible for misses. The top few (30 in this case) are responsible for the vast majority of misses. This data immediately suggests the benefit of caching and prefetching algorithms which are based on PC and try to discover the instructions which suffer the most misses, and concentrate their effort on those portions of the program. If the top 30 missing program counters could be determined at runtime, the machine could watch for modifications to the base and index registers of those memory operations; such changes could trigger a prefetch.

In the first column of Table 6.3, the list of the top 30 utilized memory addresses and the top 30 missing addresses are compared and the number of common entries is shown. It is not often that a commonly-used address also commonly misses the L1 data cache. The second column is similar but compares the list of PCs that are responsible for the most memory accesses and the list of PCs that are responsible for the most misses. It is evident that PCs more commonly are used a lot and miss a lot. This is because a given program counter is often used to perform memory operations to many different addresses. Both columns are for the best112 configuration.

6.5.4. Correlation of Data Address and PC with Access and Misses

Figure 6.14 summarizes some of the data shown in the previous two sections, in this case for the best112 optimized benchmarks. The first column is taken from the best112 column in Figure 6.5; the second column is taken from Figure 6.7; the third from Figure 6.10; and the fourth from Figure 6.12. The graph encodes in a more compact way the correlation between the following pairs: data address and access; data address and miss; program counter and access; and program counter and miss.

The data address is not very correlated at all with miss activity. However, the data address does tell us more about the access patterns of the program, as heavily accessed data addresses are responsible for a significant portion of accesses.

The program counter is generally more highly correlated to both accesses and misses than the data address. Knowing the top few commonly missing program counters covers a huge portion of the miss activity of the program.

These graphs give a clear indication that the characteristics of the load or store instruction themselves are more important than the characteristics of the data address being accessed. Cache misses, for example, are spread about the address space but are concentrated with respect to the program counter. The program location of these memory instructions and their form give much more context as to what might happen than the address which they access. Actually, the instruction and its location tell us something about the data that is accessed.

Once the subset of program locations that are critical to memory access and miss activity is known, then optimizations can be focused on those location. Unfortunately, the difficulty at that point is making a determination of what data address is accessed. This is the trouble with prefetching, for example. We can determine, with the data above, which subset of the load instructions require prefetching. However, it is difficult to determine the address to prefetch because each such missing program counter is responsible for touching a large number of addresses. This is proven in the correlation statistics and also by our

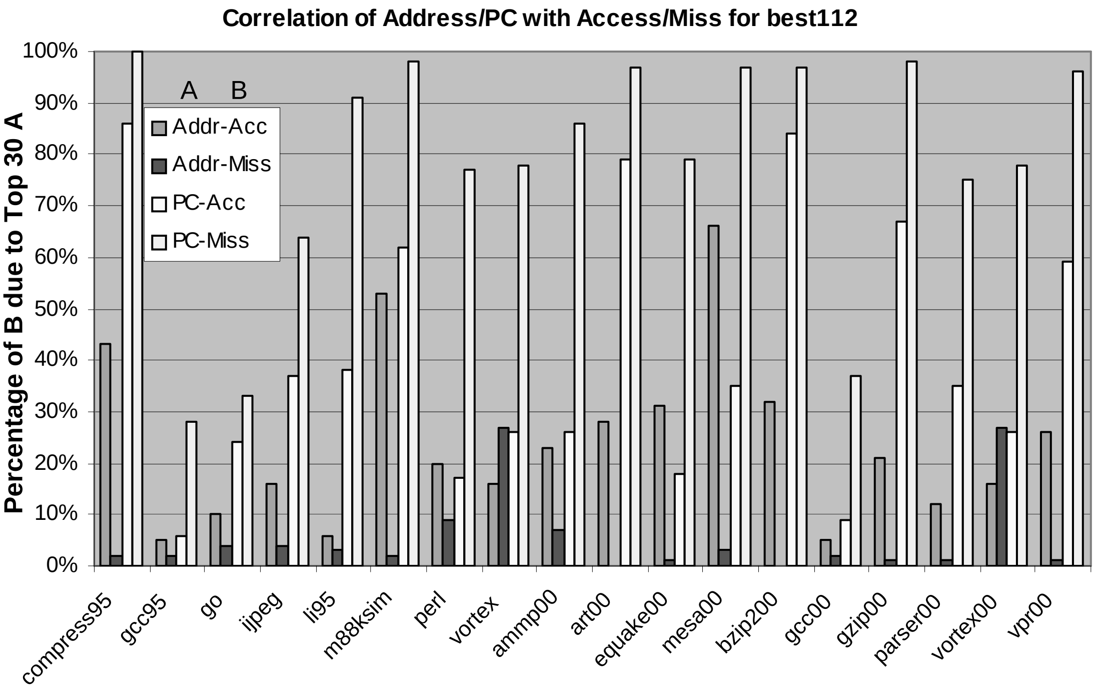


Figure 6.14: Correlation of data address and PC to access and misses.

experience with programs–the load instructions that tend to miss are those that walk arrays or dynamically linked structures such as lists and trees.

The correlation of data address and program counter and program counter is further illustrated in Table 6.3. This table shows, in the first column, the number of data addresses which both are frequently accessed and frequently missed. This is found simply by finding the common entries in the top 30 addresses that cause misses and the top 30 addresses that are responsible for memory activity. In most cases, very few addresses are found on both lists.

On the other hand, a number of benchmarks have a significant number of program counters which are responsible for both many accesses and many misses. This is shown in the second column.

6.5.5. Source Code Examples

While the data above is interesting from an abstract standpoint, it leaves something to be desired as far as specifying exactly what is happening. For this reason, I have selected a number of benchmarks and examined them in more detail to determine where

Benchmark	Data Addresses	PCs
compress95	0	13
gcc95	7	11
go	2	9
ijpeg	0	14
li95	0	5
m88ksim	0	6
perl	0	3
vortex	0	6
ammp	0	4
art	0	15
equake	4	1
mesa	0	0
bzip	0	2
gcc00	10	15
gzip	0	11
mcf	0	0
parser	4	6
vortex00	0	6
vpr	2	7

Table 6.3: Data addresses and instructions which both access and miss often.

the memory accesses and misses are occurring. It is important to do this from the perspective of the source code so that we can learn something about basic program characteristics that lead to the memory behavior that we have observed in the previous sections.

For the `art` benchmark, all the memory activity comes from the function `train_match` in `scanner.c`. The important snippet is shown in Figure 6.15. The code consists of three double-precision floating point loads followed by some arithmetic and a store. The first two floating point loads are responsible for the most misses in the benchmark, and overall those two plus the other floating point load and store are responsible for the most accesses. The latter two can be removed with register promotion because the address is invariant in the innermost loop body.

The `m88ksim` benchmark shows somewhat different behavior. The most accessed memory locations come from pointer-based loads and stores in the `alignd()` function in

```
397:  while (!matched)
398:  {
399:    f1res = 0;
400:    for (j=0;j<9 && !f1res ;j++)
401:    {
....    ....
470:      /* Compute F2 - y values */
471:      for (tj=spot;tj<numf2s;tj++)
472:      {
473:        Y[tj].y = 0;
474:        if ( !Y[tj].reset )
475:        for (ti=0;ti<numf1s;ti++)
476:           Y[tj].y += f1_layer[ti].P * bus[ti][tj];
477:      }
....    ....

$L282:  # Assembly code for loop around line 476 in C source
       lw $3,($13)
       l.d $f4,40($14)    # accessed and missed a lot
       l.d $f2,($11+$3)   # accessed and missed a lot
       l.d $f0,0($6)      # accessed a lot
       mul.d $f2,$f4,$f2
       add.d $f0,$f0,$f2
       s.d $f0,0($6)      # accessed a lot
       addu $13,$13,4
       addu $14,$14,64
       bltu $14,$10,$L282
```

Figure 6.15: Code in art that is responsible for accesses and misses.

alignd.c. The pointers are passed into the function and the contents of memory at those locations is modified. These data items are impossible to allocate to registers. A similar instruction sequence from line 33-34 of that source code behaves similarly.

Nevertheless, these operations do not account for a large percentage of the cache misses. Instead, pointer chasing and hash table lookups generate the most misses, as shown in Figure 6.17. Here, the array lookup in hashtab is randomized because of the hashing function; once the value for ptr is established, some dynamically linked structure is traversed. Both of these situation are difficult to handle because the address is computed immediately before the load is performed; in the one case, because of a hashing calculation; in the other, because one load depends on the previous load in a tight loop.

```
41:      for (*s = 0 ; expdiff > 0 ; expdiff--) {
42:         *s |= *amantlo & 1;
43:         *amantlo >>= 1;
44:         *amantlo |= *amanthi<<31;
45:         *amanthi >>= 1;
46:      }

        # Assembly code for line 43-44 of C source
        sw $3,($6)   # accessed a lot
        lw $2,($5)   # accessed a lot
        sll $2,$2,31
        or $2,$3,$2
        sw $2,($6)   # accessed a lot
```

Figure 6.16: Code in m88ksim that is responsible for memory activity.

```
43:   ptr = hashtab[key % HASHVAL];
44:
45:   while (ptr != NULL && ptr->opcode != key)
46:      ptr = ptr->next;

        # Assembly code for lines 43-46 of C source
        li $2,79
        remu $2,$4,$2
        sll $3,$2,2
        la $2,hashtab
        lw $2,($2+$3)    # missed a lot
        beq $2,$0,$L9
$L10:
        lw $3,0($2)
        beq $3,$4,$L9
$L11:
        lw $2,40($2)
        beq $2,$0,$L9
$L13:
        lw $3,0($2)      # missed a lot
        bne $3,$4,$L11
```

Figure 6.17: Code in m88ksim that is responsible for misses.

Neither of these can be handled easily by the compiler as far as register allocation or even instruction scheduling.

```
getbyte()
{
    if( InCnt > 0 ) {
        InCnt--;
        return( (unsigned int)*InBuff++ );
    } else {
        return( -1 );
    }
}

putbyte( c )
char c;
{
    *OutBuff++ = c;
}
```

Figure 6.18: Code in compress that is responsible for memory activity.

The major contributors to memory activity in the compress benchmark are the `getbyte` and `putbyte` functions. These are shown in Figure 6.18. The assembly code is omitted since it is simple load and store operations of the `InBuff` and `OutBuff` pointers as well as load and store operations of the character `c`. The loads of the pointers themselves are a significant source of accesses; these can be removed by allocating them to registers using global variable register allocation (as in Chapter 4 and our best112 optimizations). The array stores of the characters `c` cannot be eliminated.

The most misses in the compress are due to the code shown in Figure 6.19. It is another hash table lookup based on a complicated computed address; the load is shown in boldface type. There is not much the compiler can do here either. The array is `htab`, an array of 69001 32-bit integers. This will not fit into a 256KB cache.

The final benchmark that we will present here is `go`. The primary contributor of memory accesses is the `mrglist` function in `g2list.c`. In `go`, linked list are implemented in the `links[]` and `list[]` arrays. This is shown in Figure 6.20. In this case, three load instructions are most responsible for memory accesses. The C source responsible for the loads is shown in boldface type in the figure.

The code in `go` that is responsible for the most misses is also related to list management. In `g2list.c` in the `addlist()` function, there is a pointer load that is imme-

```
fcode = (long) (((long) c << maxbits) + ent);
i = ((c << hshift) ^ ent);   /* xor hashing */

if ( htabof (i) == fcode ) {
```

```
# Assembly code for C source
lw/7:0(20) $2,in_count
addu $2,$2,1
sw/7:0(19) $2,in_count
lw/7:0(20) $2,maxbits
sll $3,$19,$2
sll $2,$19,$22
xor $18,$2,$21
sll $2,$18,2
la $4,htab
addu $20,$4,$2
addu $17,$3,$21
lw/7:0(28) $2,($20)    # missed a lot
bne $2,$17,$L68
```

Figure 6.19: Code in compress that is responsible for cache misses.

```
ptr2 = *list2;

while(ptr1 != EOL){

   /* guaranteed that list[ptr1] > list[ptr2] */

  if(links[ptr2] == EOL){         /* end case */
     links[ptr2] = freelist;
....
  temp2 = list[links[ptr2]];
```

Figure 6.20: Code in go that is responsible for memory accesses.

diately used as the source of a load out of the `list[]` array. This is shown in Figure 6.21 Again, this cannot be optimized away by the compiler. These are the top two loads responsible for cache misses. Similar constructs show up in several places in the list management code.

Table 6.4 shows, for O1 optimization, the functions in SPEC95 which are responsible for cache misses. The second column shows the function that accounts for the most

```
int addlist(int value, int* head){
  register int ptr, optr;

 if(list[*head] > value) { /* add to front of list */

  # Assembly code for C source
  la $7,list
  lw $10,($5)
  sll $2,$10,2
  addu $2,$7,$2
  lw $2,($2)
  ble $2,$4,$L96
```

Figure 6.21: Code in go that is responsible for cache misses.

Benchmark	Function that Misses L1 D cache the most	Percent Misses
compress	getbyte	45%
gcc	bzero	7%
go	findshapes	10%
ijpeg	rgb_ycc_convert	24%
li95	sweep	27%
m88ksim	memset	47%
perl	_wordcopy_fwd_aligned	18%
vortex	Chunk_Validate	26%

Table 6.4: The worst missing functions in SPECint95.

L1 data cache misses and the third column shows the percentage of misses that is. The simulations are with a 16KB cache. The data shown are essentially the same for both the Mirv and gcc compilers; they are always within 1-2% of each other and the same function is always responsible for the most misses in both cases. Most of the benchmarks, like compress, ijpeg, li95, m88ksim, and vortex are have heavy weight on the top-most function. Interestingly, in those cases the other misses tend to be clustered in the next few functions. In those cases like gcc, the misses are well-distributed among a number of functions. In any case, the top functions should be the target of compiler and hardware optimizations.

Several conclusions can be drawn from this analysis. First, very complicated memory instructions are typically responsible for accesses and misses. Second, memory accesses and misses often occur in bunches. Short snippets of source code often contain two or more memory operations which are responsible for a lot of accesses or misses. This makes it difficult to optimize these cases, especially sine they are often dependent on each other.

6.6. Conclusions

This chapter has shown how optimizations affect the cache performance of the benchmarks studied. Two new metrics were introduced to characterize the optimizations relative to the hardware reductions that they could sustain and still achieve similar performance levels compared to normally-optimized binaries on the original hardware.

The chapter then examined a number of metrics in order to determine the characteristics of memory operations in the benchmarks. We determined that the optimizations were effectively caching heavily-used values into the register file. The optimizations are localizing accesses and misses to fewer program counters, but those memory operations are generally in loops and address a wide range of memory locations so that they would be difficult to optimize. This tends to manifest those memory operations that are tough to handle, simply because they could not be easily optimized. In other words, compiler optimization is handling those items which are somewhat easy for the hardware to cache. This is good because putting them into registers makes it that much faster for the common case. On the other hand, this leaves a more difficult job for the hardware because it filters many of the “easy-to-cache” memory operations and leaves the remaining operations.

There are a number of ways in which this work could be extended. We have introduced the IWE and CLE metrics for characterizing the value of compiler optimization relative to hardware complexity. There are many other hardware structures that could be studied in like manner. Since the optimizations remove instructions, particularly loads and stores, the window size, the load-store queue size, the number of L1 data cache ports, the number of function units, etc. are all candidates for reduction in the presence of compiler optimization. The optimizations also eliminate memory operations, suggesting that a trade-off might be made in the size of the caches in addition to the latencies of them. This is a generally understood principle of trading off hardware complexity at the expense of compiler complexity, though it is difficult to study in most research environments because binary compatibility is of utmost concern, so the compiler is not modified. Our research has both the compiler and architecture at its disposal. We will discuss the compiler we used for this research in the next chapter.

Chapter 7

Design and Engineering of the MIRV C Compiler

7.1. Introduction

The MIRV Compiler [MIRV] is an experimental C language compiler. This chapter discusses the compiler and related tools which have been developed in order to support the research presented in this dissertation.

The MIRV compiler is based on a high-level prefix intermediate representation (IR), called the MIRV language, or just MIRV for short[1]. This IR allows very high-level transformations to the program code, essentially at source code level. This form is similar to that used in SUIF [Hall96]. When printed, the MIRV tree is emitted in a prefix linear form, where the context of the prefix operators allows efficient code generation in a single pass or more sophisticated code generation with register allocation and optimizations. The compiler currently compiles the SPEC95 integer benchmarks, most SPEC2000 benchmarks, and many other benchmarks and regression tests.

MIRV was designed as a research platform to support research into both high-performance and embedded computing. In addition to the studies on registers and memory optimizations presented in this dissertation, it has been used in hardware-assisted compiler optimizations and in code compression studies. MIRV was also designed so that optimization filters would be easy to code and isolated from the rest of the system as much as possible. MIRV was also designed to be extensible through intermediate representation annotation fields. These fields can be used to pass information from the frontend to the backend or the simulator. Examples of such information include annotations for statistics gathering and operation modifiers to cause the backend to use different opcodes. These

[1] Note that the term “MIRV” is overloaded: it refers to the compiler or the IR, depending on the context.

same annotations are used internally in the compiler to pass information between optimization phases. MIRV was also designed in a decoupled way that would allow easy retargeting of the backend. The compiler has already been targeted to three architectures: Intel IA32, SimpleScalar/PISA, and ARM9.

The remainder of this chapter is devoted to presenting an overview of the compiler, with particular emphasis on portions used extensively in this dissertation.

7.2. History and Development Team

Originally, MIRV was conceived by Kris Flautner with ideas from Peter Bird in Peter's undergraduate compiler course. Kris shortly took to other research interests and David Greene and I continued the project. The intermediate language of MIRV was designed to be high-level and intended to be suitable for online code distribution in a just-in-time or dynamic compilation environment (similar to JAVA which later became popular). This desire necessitated that the MIRV IR could be quickly compiled to native instructions. This was one reason for the prefix intermediate form, which provides sufficient context that code can be generated using a YACC parser in one and a half passes. Unfortunately, MIRV lacked security features to protect intellectual property; it also lacked a bytecode encoding. These things relegated the compiler to a more traditional role in optimization and profiling. We used it as the basis for this and other research.

MIRV was developed with a philosophy that a small design team is best to foster communication and quickly fix bugs. For its first three years, from 1997 to 2000, the MIRV compiler development team was never above four people. For the majority of that time, only two developers were seriously working on the compiler. At the writing of this dissertation, there are only two developers, though many students in University of Michigan compiler courses have been using the tool for their class exercises.

7.3. Overall Compilation flow

Figure 7.1 shows the compilation flow of the MIRV compiler. The ovals represent files and the arrows are the compiler steps to transform the source into an executable.

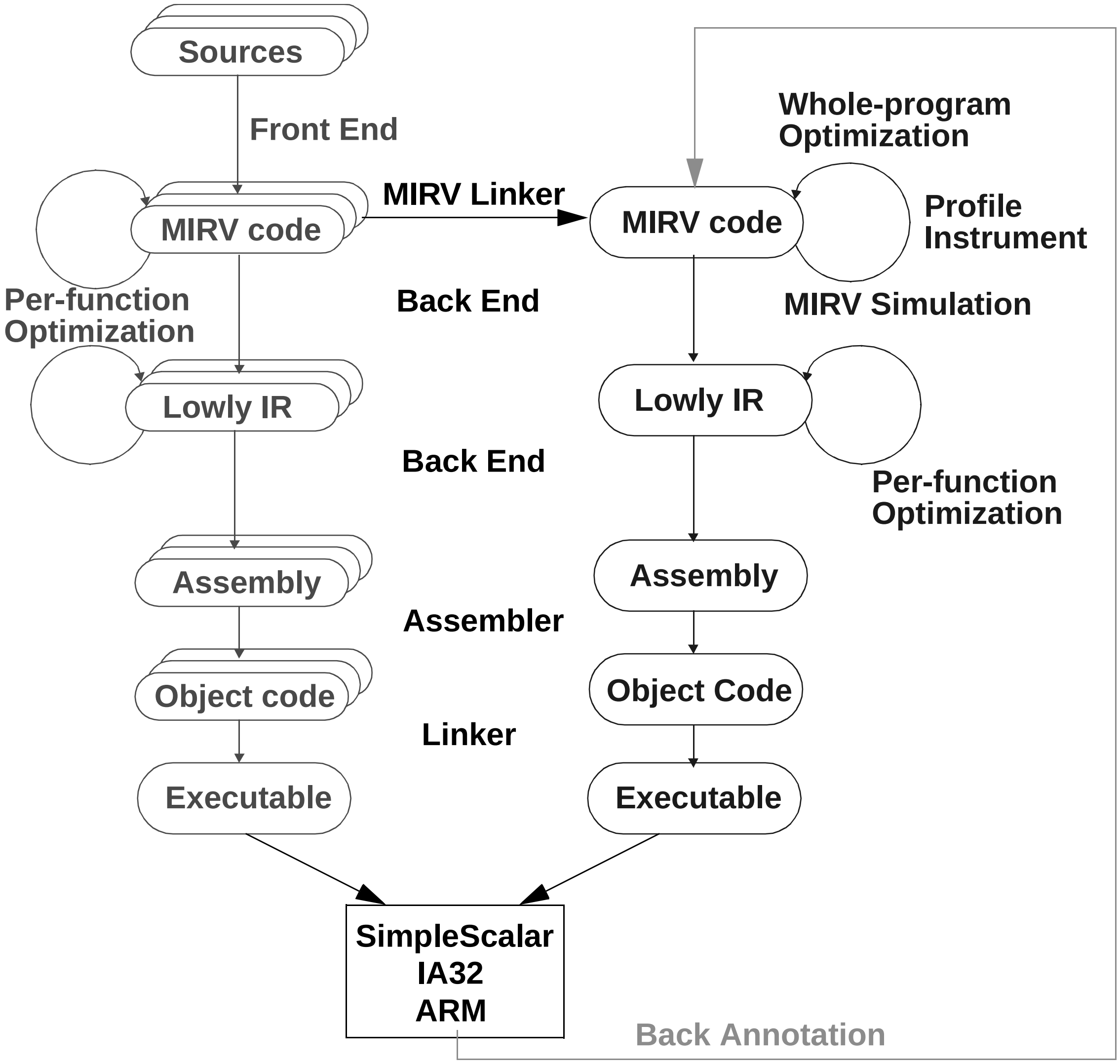


Figure 7.1: A diagram showing the overall flow of compilation in MIRV

Table 7.1 shows the order of application of both the MIRV frontend and backend filters. The steps are described in each of the following subsections.

7.4. MIRV Frontend

The MIRV compiler currently has one frontend which translates ANSI C into MIRV IR. C source is parsed by the Edison Design Group C/C++ frontend [EDG] first. After being read in by the EDG portion of the tool, the program is stored in memory in EDG's proprietary tree form. The EDG representation is slightly lower than the MIRV representation, particularly with respect to labels and `goto` statements. An IR to IR trans-

Frontend		Backend	
Optimize Level	Filter Applied	Optimize Level	Filter Applied
-O2	-fscalReplAggr	-O1	-fpeephole0
-O3	-fcallGraph	-O1	-fpeephole1
-O3	-finline	-O1	-fblockClean
-O3	-ffunctCleaner	-O1	-fcse
-O2	-floopUnroll	-O1	-fcopy_propagation
-O1	-farrayToPointer	-O1	-fconstant_propagation
-O1	-floopInversion	-O1	-fdead_code_elimination
-O1	-fconstantFold	-O1	-fpeephole0
-O1	-fpropagation	-O1	-fpeephole1
-O1	-freassociation	-O1	-fcse
-O1	-fconstantFold	-O1	-fcopy_propagation
-O1	-farithSimplify	-O1	-fconstant_propagation
-O2	-fregPromote	-O1	-fdead_code_elimination
-O1	-fdeadCode	-O1	-fpeephole0
-O1	-floopInduction	-O1	-fpeephole1
-O1	-fLICodeMotion	-O1	-flist_scheduler
-O1	-fCSE	-O0	-freg_alloc
-O1	-fpropagation	-O1	-flist_scheduler_aggressive
-O1	-fCSE	-O1	-fpeephole0
-O1	-farithSimplify	-O1	-fpeephole1
-O1	-fconstantFold	-O1	-fcselocal
-O1	-fpropagation	-O1	-fcopy_propagation
-O4	-fLICodeMotion	-O1	-fdead_code_elimination
-O1	-farithSimplify	-O1	-fpeephole1
-O1	-fconstantFold	-O1	-fblockClean
-O1	-fstrengthReduction	-O1	-fleafopt
-O2	-fscalReplAggr		
-O1	-farithSimplify		
-O1	-fdeadCode		
-O1	-fcleaner		

Table 7.1: Order of optimization filter application in MIRV.
Since the system is based on MIRV-to-MIRV filters, filters can easily be run more than once, as the table shows. The frontend filters operate on the MIRV high-level IR while the backend filters operate on a quad-type low-level IR

lation step is performed to change the EDG IR into MIRV IR. This MIRV can be written to a file or operated on in memory by the optimization filters.

Operator Class	Operator
Arithmetic	add, div, mod, mul, neg, pow, sub, sqrt
Bitwise	and, cpl (complement), or, rol, ror, shl, shr, xor
Boolean	cand, cor, eq, le, lt, ne, not, ge, gt
Casting	cast
Control flow	destAfter, destBefore, funcCall, gotoDest, if, ifElse, return
Looping	doWhile, while
Assignment	assign
Object reference	cref - specifies a constant value vref - specifies a variable object deref - specifies a reference through pointer aref, airef - array access through direct object or pointer vfref vfiref - field access through direct object or pointer
Object size	sizeOf
Symbol Table	tsdecl, tcdecl, tfdecl, tudecl, cdecl, fdecl, vdecl

Table 7.2: The MIRV high-level IR operators.

7.5. The MIRV High Level Intermediate Language

The MIRV IR's operators are summarized in Table 7.2. This section will describe the syntax of these operators through a sequence of examples.

7.5.1. Basic Symbol Information: Unids

Symbols in MIRV are uniquely identified with what is called a *unid*. A unid is simply a "unique identifier". In the present MIRV implementation, this consists of a string. It could be a number (in fact, it formerly was) or any unique identifier. The unid manager maintains the mappings from string to unid and ensures that no unid is used twice. Unids are used everywhere in MIRV: for types symbols, global and local variables, functions, packages, modules, labels, and goto statements. MIRV has a single-level namespace which is implemented in two symbol tables for efficiency: one for the global scope, and one for the function scope. It is the responsibility of the language frontend to flatten the source namespace into this form.

```
package version 0 0 1  unid __mirv_pack version  0 0 0 {
   name "add.mrv"
}

module unid __mirv_pack.m1 {
  ...
```

Figure 7.2: Example package and module declarations.

7.5.2. Packaging Constructs

Every MIRV file begins with a package and module declaration. Examples are shown in Figure 7.2. The package declaration is essentially superfluous at the present, but was intended to contain version information on the code so that (in a just-in-time compilation environment) the compiler could select the appropriate version of a package. The package has two versions and a textual name.

The module declaration is essentially similar to a C source file. It contains the global scope of the file and all the types, variables, and functions in the source file. It is designated with a unid, in our case, always `__mirv_pack.m1`.

7.5.3. Simple Type Declarations

Every MIRV file contains the complete symbol table needed to process the file. There are no built-in or “understood” types. A few examples of the symbol table syntax in MIRV are shown in Figure 7.3. Each simple type declaration begins with a `tsdecl` (type-simple-declaration) keyword. In the first case, a 32-bit signed integer type is being defined. The unid sint32 is the unique handle for this type. By convention, simple integral types are named similarly to this one: the first letter is `s` or `u`, for signed or unsigned, then `int`, then the size, in bits, of the type. After the unid, the declaration explicitly describes the type with the text `integer signed 32`. The second declaration shown in the figure is of a C `char*`. It is called `sint8p`, for a pointer to a signed integer of size 8. The `attribute` blocks before each declaration simply specify a textual name for the type. This is not strictly necessary in MIRV either but was useful previously when unids were

```
attribute {
   name "sint32"
}
tsdecl unid sint32
   integer signed 32

attribute {
   name      "sint8p"
}
tsdecl unid sint8p
   pointer unid sint8        # sint8p
```

Figure 7.3: Example simple type declarations.

not encoded in a such a readable form. Attributes will be discussed more fully in Section 7.5.13.

Unids can have any name in MIRV as long as they are consistent. We chose the name “sint32,” for example, so the MIRV code is easy to read.

7.5.4. Complex Type Declarations

Three examples of complex type declarations are shown in Figure 7.4. The first is a structure called Z which has two fields, x and y. The structure is declared with a tcdecl keyword (type-complex-declaration). It has a unid struct_Z, and then two field specifiers tfield. The fields are given unids, Z.x and Z.y, and their types are specified by the unid sint32.

The second type declaration is of a union called u. It is declared with the tudecl keyword (type-union-declaration). It also has two fields, c and i. The second field, i, is a sint32 as in the previous example. The first field is of type sint8_4_, which is a char[4] in C jargon.

The third example declares a function type. In this case, it is the declaration for the type signature of the printf function, which takes a char*, a variable argument list, and returns an integer. The name attribute shows this, with the sint32 return value shown first, then an open parenthesis, then the sint8* (equivalent to sint8p, described

```
attribute {
   name      "struct_Z"
}
tcdecl unid struct_Z {       # struct_Z
   tfield unid Z.x unid sint32       # sint32 Z.x
   tfield unid Z.y unid sint32       # sint32 Z.y
}

attribute {
   name      "union_u"
}
tudecl unid union_u {        # union_u
   tfield unid u.c unid sint8_4_     # sint8_4_ u.c
   tfield unid u.i unid sint32       # sint32 u.i
}

attribute {
   name "sint32(sint8*, ...)"
}
tfdecl unid sint32_sint8p__val_ {
   retValType unid sint32
   argType unid sint8p
   varArgList
}
```

Figure 7.4: Example complex type declarations.

above), and then a "`...`" to indicate the variable argument list. The declaration begins with a `tfdecl` keyword (type-function-declaration). It is followed by a unid for the type which is similar in format to the name attribute, except that instead of "`...`", the keyword `val` is used. This is done because periods are not allowed inside the name of a unid except for field specifiers in structures and unions, but "`...`" is easier to read in the name attribute. The declaration then explicitly indicates that the return type (`retValType`) is a `sint32`, the first argument type (`argType`) is an `sint8p`, and the remainder of the arguments come in a `varArgList`.

7.5.5. Function Declarations

Function declarations for functions always appear in MIRV before the function's definition. The definition of the function contains its body while the declaration is just a

```
fdecl import unid printf unid sint32_sint8p__val_

attribute {
   name "main"
}
fdecl export unid main unid sint32_sint32__sint8pp__sint8pp_

fdecl internal unid foo unid sint32_sint32
```

Figure 7.5: Example function declarations.

forward indication of the function so that function call statements can use it without the body having yet been seen. Two example function declarations are shown in Figure 7.5. The first declares the printf function with the type `sint32_sint8p__val_` that was described in the last section. The function is declared with the `fdecl` (function-declaration) keyword. The following `import` keyword declares that the function is from another module and will not be defined in the present module. The unid for this function is simply the name of the function, `printf`, and the type is given last.

The second example is a declaration of the function `main`. In this case, `main` is declared `export` because it is going to be defined in the present module and is available for other modules to `import`.

The final example illustrates the other scoping keyword in MIRV. This declaration is of the static function called `foo` (`static int foo(int)` in C parlance). The `internal` keyword in the declaration indicates that the function is going to be defined in the present module but should not be made available for other modules to `import`.

7.5.6. Constant Declarations

Constant declarations are very similar to the declarations which we have already seen. The `cdecl` keyword is used to introduce the declaration. The unid for the constant symbol appears next. In the first example of Figure 7.6, the unid is `sint32_8`, which is the conventional name for a 32-bit signed integer with the value 8. The type unid (`sint32`) is given next, and finally the integer value.

The second example declares a constant string “`Hello, World!\n`”. The unid in this case, since it is an compiler-generated string constant is given the unid `mod-`

```
cdecl unid sint32_8 unid sint32 8

cdecl unid module.m1.25 unid sint8_15_ "Hello, World!\n\0"
```

Figure 7.6: Example constant declarations.

```
fdef unid main {
  ...
  return
}
```

Figure 7.7: Example function definition.

ule.m1.25. The type of this constant is sint8_15_, which is a char[15]. The string has 14 characters plus one for the trailing NULL byte. Finally, the value of the constant is given. The newline and NULL byte are encoded explicitly in the MIRV.

7.5.7. Function Definitions

An example functions is shown in Figure 7.7. It is simply specified with a fdef (function-definition) keyword and the function's unid (as previously defined). The body of the function ("...") is a list of MIRV statements (see Section 7.5.9). The function ends with a return statement.

7.5.8. Global and Local Variable Declarations

Several examples of variable declarations are shown in Figure 7.8. The first example is a variable declaration (vdecl) of a global variable called j. The variable is declared export, meaning it is externally visible, and is of type sint32. It has an initializer given by the init keyword whose value is 8. The constant is referred to by a cref keyword. This is the same constant as declared in Figure 7.6.

```
attribute {
  name "j"
  register false
  used true
}
vdecl export unid j unid sint32 {
  init { cref unid sint32_8 }
}

fdef unid main {
{
  retDecl unid main.__return_value unid sint32


  argDecl unid main.__parameter_1_ unid sint32


  vdecl internal unid main._6_7_k unid sint32 { }
```

Figure 7.8: Example global and local variable declarations.

The second example, inside of the definition of function main, is a `retDecl`. This declaration declares a variable whose unid is `main.__return_value` of type `sint32`. This is a special variable where the function return value is assigned; the back-end allocates this variable to the appropriate machine location (memory or register) according to the application binary interface. Argument variables are declared similarly, but with the `argDecl` keyword.

Finally, functions can declare local variables with the `vdecl` keyword, as described above. In this case, a variable called `k` is declared, but there is no initializer for this variable. The unid of local variables (in this case `main._6_7_k`) is made up of several components. The first is the function name. The last is the variable name. The middle numbers indicate the row and column number of the C source file where the variable is declared. This information is useful for debugging.

```
assign
          vref unid main._7_7_i
          +
             vref unid main._7_7_i
             cref unid sint32_1

fcall unid printf
         {
            addrOf
               unid sint8p
               cref unid p.m1.1 # i should be %d, it is
%d\n\0
            vref unid main._6_7_k
            vref unid main._7_7_i
```

Figure 7.9: Example statements.

7.5.9. Statements

Two simple statements are shown in Figure 7.9. The first example in that figure shows how the C code `i = i + 1` is implemented. The variable `i` is referenced by the `vref` keyword twice in that statement. The + operator indicates the addition. (This operator can also be specified textually as `add`.) Finally, the constant 1 is indicated by the `cref` keyword.

The second example shows how a function call is specified in MIRV. The `fcall` keyword is followed by the unid of the function which is being called. A brace-delimited block of arguments is then specified. In this case, a call is being made to `printf("i should be %d, it is %d\n", k, i)`. The first parameter is the address of (`addrof`) the constant string. The second and third parameters are simple `vref`s of `k` and `i`.

Notice that function calls cannot be nested and must be the immediate RHS of an assignment or a call statement. This restriction serves to isolate the variety of locations where side-effects are an issue, and simplifies processing of function calls in the later optimization and code generation steps.

```
  while
            <                               # LOOP EXIT
CONDITION
               vref unid main._7_7_i
               vref unid main._6_7_k
            {
               ...                          # LOOP BODY
```

Figure 7.10: Example looping constructs.

Because of the large variety of operators in MIRV, we will not shown an example for each. The full set of operators is shown in Table 7.2 and examples are available online [MIRV].

7.5.10. Looping Constructs

MIRV has two primary looping constructs: `while` and `doWhile`. The example in Figure 7.10 shows a `while` loop. The loop is controlled by an less-than expression which compares variables `k` and `i`. The body of the loop is a list of statements (specified as “`...`” in the example.)

7.5.11. Simple Control Constructs

Figure 7.11 shows examples of MIRV’s simple control structures. The first is an `ifElse`, which is specified in a manner similar to C. One difference is that the condition expression controlling entrance into the if body must not have side effects. In general, MIRV expressions cannot have side-effects: assignments that are implicit in C (such as pre- and post-increment) are specified explicitly in MIRV. An `if` statement (without an else) is specified with the `if` keyword. The `if` and `else` bodies are themselves lists of statements (specified in this example by “`...`”).

The second example shows a `switch` statement. The variable `X` is the switch control. The `switch` syntax then requires a low and high numeric specifier indicating the range of the case statement. The keyword `default` is supplied to indicate early in the switch statement that any un-referenced labels should reference the default case. This

```
ifElse
  ==
    vref unid main._7_7_i
    cref unid sint32_2
   {
     ...                             # IF BODY
   }
   {
     ...                             # ELSE BODY
   }


switch
          vref unid main._1553_7_X
               1 4 default {         # MIN MAX (DEFAULT)?
          case { 1 }
             { ... }
  case { 2 3 4 }
             { ... }
  case default
             { ... }
}
```

Figure 7.11: Example simple control constructs.

design is one demonstration of the utility of the prefix nature of the MIRV language. The key information is specified at the top of the construct so that appropriate context (such as a label vector in this case) can be set up before the remainder of the `switch` statement is parsed. The remainder of the `switch` code is devoted to the specification of the cases. Each case is specified by the `case` keyword and can have multiple cases per keyword. Each case has an associated block of statements with it.

7.5.12. Complex Control Structures

MIRV has support for complex control structures as well. Arbitrary C `goto` statements are supported as well as the `break` and `continue` statements. Figure 7.12 shows how MIRV implements the C `continue` statement. Assume that the whole body of code shown in the figure is contained in a `while` loop. The while loop contains a single statement, called a `destAfter`. The `destAfter` specifies a label, `main.0` in this case,

```
destAfter unid main.0
{
 ...
 if
   ==
     vref unid main._7_7_i
     cref unid sint32_2
  {
    gotoDest unid main.0          # IMPLEMENT CONTINUE
  }
 ...
}
assign
 vref unid main._7_7_i
 +
    vref unid main._7_7_i
    cref unid sint32_1
```

Figure 7.12: Example complex control structures.

which later code uses to refer to this `destAfter` statement. The `destAfter` contains a block of arbitrary MIRV code. Inside that block, there is a `gotoDest` statement with a label unid. The semantics of the `gotoDest` are simple: jump to the point *after* the specified `destAfter` statement. Thus, control flow is redirected to the `assign` statement.

The `destAfter` (and corresponding `destBefore`) is a structured way to specify complex control flow. Because MIRV is designed primarily as a structured high-level intermediate form, it is convenient to be able to specify as much control flow as possible in a structured way. MIRV has a filter to convert C `goto` statements to structured control flow whenever possible.

7.5.13. Attributes

Attributes are used to extend the MIRV language without changing the basic grammar. An attribute can be an int, bool, double, string, or an internal user-declared class. All the attributes that MIRV currently supports are detailed in Table 7.3.

There are several other attributes, not documented here, which are for internal use only.

Type	Name	Description
String	Name	A textual name for a symbol.
Integer	Line	The source line number of a statement.
	InvokeCount	The number of times a function was invoked.
	switch_GCD	Marker for an optimized switch statement.
	switch_shift	Marker for an optimized switch statement.
	FrequencyCount	A static estimate of how often a variable is used.
	BlkNum	A unique identifier for a MIRV block.
	BlkCnt	The dynamic number of times that a block was executed.
	DynFreqCnt	A dynamic count of how often a variable is used.
	MaxLive	How many variables are alive at one time in a function.
	StartAddress	Starting address of function in executable representation; used by profiling to account for time spent in functions.
	EndAddress	Ending address of a function.
Boolean	Register	Whether a variable can be enregistered.
	Used	Whether a variable is used or not.
	BlkPrf	Marker on block profiling instrumentation code that allows it to be deleted later.
	Unstructured	Whether a function contains unstructured control flow (gotos).
	Leaf	Whether a function is a static leaf (has no function call sites).
	UnrolledLoop	Whether the loop was generated by an unrolling operation.
	MirvROData	Whether a datum is read only.
	Replaced	Marker that ensures propagation does not propagate something it just propagated without redoing dataflow analysis.
	Rematerialize	Marker to avoid undoing CSE in propagation.
	NotPromoted	This reference was considered as a promotion candidate but was not promoted.
	SpecPromote	This assignment is a speculative promotion operation. Used to generate special load opcode.
	SpecUse	This reference is a use of a speculatively promoted value.

Table 7.3: MIRV attributes.

Type	Name	Description
	SpecDemote	This assignment is the speculative demotion of a previously speculatively promoted value. Used to generate special store opcode.
	Promote	This assignment is a promoting load operation.
	Demote	This assignment is a demoting store operation.
Double	ExecTime	Execution time for a function
	CumulativeExecTime	Execution time for a function and all of its children.
Node	AliasDataflow	Collection of alias sets at each statement. An alias set is a mapping of pointers to the data that they potentially point to.
	Alias	For each reference to a data, it is the point-to set for that data.
	AliasRef	All the data that can be referenced by a called function.
	Data	Abstract name for a data object.
	Def	Set of data that is defined by a statement.
	Use	Set of data that is used by a reference.
	DefUseChain	Marker that indicates whether reaching-definition analysis was run.
	DefUseSummary	Mod-ref information for a function. It contains the non-local data items which are potentially modified or referenced by the function.
	ReachingDef	For each statement, collection of definitions that reach the statement.
	Replacement	For each node on the parse tree, provides a back pointer to the parent node and implements node self-replacement.
	AvailableExpression	For each statement, the collection of expressions that reach the statement.
	ValueProfile	For each function, all the parameters and the values they take.
	Labelflow	Store goto and label information.
	LiveOut, LiveIn, LiveVariable	Used during inlining to estimate when inlining should not be performed because of high register pressure.

Table 7.3: MIRV attributes.

7.5.14. Potential MIRV IR Modifications

Throughout the development of MIRV, several changes have been made to the original IR design. This section summarizes a number of those and discusses what could be changed in the future.

Past Changes

1. Unids were numeric. This was easy in the code but made debugging difficult. They were then changed to a hierarchical name with a list of identifiers to specify the unid. This was too expensive. Presently, the unid is simply a string. This provides the best trade-off in execution time and readability.
2. Type sizes were specified with some flexibility. The frontend could `suggest` or `require` that a type have a certain size. We eliminated this to simplify the language. Type sizes are all required sizes. The backend can implement a type with whatever size it likes, as long as it guarantees compatibility with the type specified in the MIRV.
3. Early versions of MIRV did not support arbitrary `goto` statements. This was intended to simplify dataflow analysis so that only structured code had to be supported. However, so many benchmarks use `goto`s that this restriction had to be removed. Functions with gotos in them may be converted to structured code or simply not optimized, dependent on the optimization filter that is running. Previous research and experience with MIRV show that `goto`s usually can be eliminated in favor of more structured approaches.
4. Early versions of MIRV had an `offsetof` operator which, given a field, would return the offset of that field from the beginning of the structure. In the present version, this is replaced by the `addrof struct.field` construct.
5. There was no `sizeof` operator in early versions of MIRV. It was added to enhance cross-platform compilation. The backend now determines the final size of all data objects, particularly structures.
6. While and doWhile loops had initializer and pre- and post-execution blocks. These have been removed and the frontend just places those codes in the appropriate locations before or after the loop.

Future Changes

1. The package and module hierarchy could be simplified. Packages are do not serve any useful purpose at this point.
2. New source languages and cross-platform compilation issues need to be addressed more fully. Right now, MIRV is primarily designed to compile C. Other enhancements need to be made to the language in order to support C++ and other languages.
3. Code in loop conditions is sometimes duplicated in MIRV, especially if in the source language it has side effects such as function calls. This bloats the code more than necessary. Other issues related to side effects have been troublesome.
4. Evaluation of the short-circuiting operators && and || in C is implemented poorly and some changes to the language may be necessary to rectify this problem.
5. Multi-dimensional arrays are supported in MIRV. It has been debated back and forth whether these should be changed into pointer arithmetic for the backend. Currently, the transformation to pointer arithmetic is done to expose more opportunity for high-level optimization. The backend supports either.
6. There is no way to represent the (internal) node attributes in the MIRV IR. This may be useful to store information from one compilation to another.
7. No bytecode representation has been defined for MIRV.
8. MIRV has no built-in types. Changing this would allow the MIRV IR files to be smaller.
9. A number of syntactical cleanups in initializer code and elsewhere would serve to shorten the MIRV files significantly.
10. Support for volatile variables.
11. Null statements (called “none” statements) in MIRV are used too much.

7.6. MIRV High-Level Linker

An important feature of the MIRV compiler is the MIRV high-level linker. This program reads several MIRV files and links them together into a single large MIRV file.

The linked files, which are usually a whole program (minus standard libraries) can then be optimized together. This allows whole-program analysis and optimization. Because optimizations are often restricted simply by lack of code visibility (in a separate compilation environment), whole-program analyses and optimizations can result in significant performance improvements.

Another important consideration is the code-reuse that is provided by the linker. Many compilers must have one set of optimizations to work at the high-level IR, and another to operate on code after (low-level) linking. In MIRV, once linking is done, the code is in the same basic format as unlinked code, so that optimization filters can be used both before and after linking, without modification.

Currently MIRV implements several whole-program optimizations. One is function inlining. The linker allows the inliner to view the whole program at once, build the call graph, and decide which functions to inline based on their characteristics (how many call sites are they called from, whether they are leaf functions, etc.) The linker allows unused function to be deleted; in separate compilation, all functions must be retained because of restricted visibility.

Another whole-program optimization is global variable register allocation. Conventional compilers places global variables in memory. MIRV allows globals to be placed permanently into registers; all the references to the global are changed so that instead of referring to memory, they refer to the appropriate registers.

Whole-program analysis can be also used by the definition-use and alias analysis filters in order to do simple interprocedural side-effect analysis [Bann79].

As with any linker tool, several hurdles had to be overcome to make the MIRV linker function properly. In addition, because the linker operates on a high-level source form, there are several additional issues that must be addressed.

Global variables and functions are made to have unique names so that there are no name collisions. MIRV also contains type information in the MIRV file[2], so types also must be merged and named appropriately. Anonymous types present a difficulty because the same type can have different names in different MIRV files. The type names for basic integral and pointer types are standardized so that this is not a problem for them.

[2] The MIRV symbol table is contained in the MIRV file.

To implement merging of anonymous types, the linker must examine each struct or union type as it is encountered. If a matching type for a unid is found, the structs are compared. If they match, parsing can continue. If they differ, the newly read type must be renamed.

Since the newly-read type may have been read in under a different name, all of the structure or union types known to the linker are examined. If any matches are found, the newly-read unid is renamed to the unid of the matching type. Otherwise a new unid is generated. Such renaming also requires renaming of the field unids since the field names are based on the name of their containing type. Any types using the renamed unid (it may have been forward-declared) will need fixing-up to point to the correct type. These will also likely be renamed since their unids are generated based on their child types.

Note that such mappings are only valid for a single source file. A new source file will have different unid names for its types and requires a new mapping table. This renaming could be avoided by standardizing names for structure and union types. This is nontrivial as all of the field information in the type would need to be encoded in such a naming scheme, including nested structure types.

7.7. MIRV Filters

The "middle end" of the MIRV compiler consists of many "filters" which collect information about the code and optimize it. The filters are designed with a modular interface so that they are easy to write. The compiler is written in C++. Well-known programming "patterns" such as the visitor, singleton, and factory are used throughout the compiler [Gamm95].

The filters currently implemented in MIRV are show in Table 7.4. The second column indicates whether the filter is for transformation (T), analysis (A), or instrumentation (I). An analysis filter collects and propagates information about the source program. The most common example is the definition-use (defUse) filter. It performs reaching definition analysis. It in turn uses the alias analysis filter. The register promote filter is an example of a transformation filter. It uses the defUse information, among other analyses, to determine when it can promote a memory-based variable to a register over a loop region. It then does the appropriate transformations to the code. An instrumentation filter adds MIRV code or other attributes to the MIRV tree in order to allow later passes or simulations to collect information. The block profile filter is an example of an instrumentation filter. These filters will be described later in this report.

7.7.1. The Register Promote Filter

The following is a simplified explanation of how the algorithm proceeds. Each function in the module is treated separately, except for the alias information that is shared between functions.

1. Collect global scalar variables and constants into *globalCandidateList*
2. Collect aliased scalar local variables into *localCandidateList*
3. Walk through the function and find each variable or constant reference, or suitable dereference. If it is a direct reference is to a *globalCandidate* or *localCandidate*, store this reference into the *candidateReferenceList*. There is a *candidateReferenceList* for each promotion region. If it is a suitable dereference, also add it to the *candidateReferenceList.*
4. For each promotion region, remove aliased candidate references from the *candidateReferenceList*. An alias exists if the alias analyzer has determined there is a potential definition or use of the candidate which is not through the candidate's primary name.
5. For each promotion region, promote all candidates that remain in the *candidateReferenceList* for the region. Do not promote something in an inner loop if it can be promoted in an outer loop.

In MIRV, dereferences are only promoted in loop regions. A “suitable dereference” is a load or store operation which satisfies a number of conditions. For this work, these are:

1. The base address is a simple variable reference.
2. The dereference is at the top level block of the loop, i.e. not in a conditional clause. Otherwise promoting it might cause a fault.
3. The loop must be a do-while loop so that it is guaranteed to execute at least once. This is often satisfied because MIRV performs loop inversion to reduce

Filter		Description
CSE	T	Traditional common-subexpression elimination.
LICM	T	Traditional loop-invariant code motion.
Arithmetic simplification	T	Simplifies computations like x + 0 to x.
Array to pointer	T	Convert array references to explicit pointer arithmetic.
Block Profile	I	Add a counter array and counter increment statements in each block statement in the program.
Call graph	A	Gathers function call information and creates call graph.
Cleaner	T	Flattens nested block statements.
Constant fold	T	Evaluates expressions whose operands are constant.
Constant propagate	T	Propagate constants through assignments to later uses.
Copy propagate	T	Propagates variable through a copy to later uses.
Dead code removal	T	Delete dead code from MIRV tree.
Definition-use	A	Traditional reaching definition analysis. Marks the definitions that reach a use.
Frequency count	A	Mark each variable with a count indicating how many times it is used or defined in the program. This can be done statically by estimating loop trip count or dynamically if the block profiler has been used to instrument the program.
Function inline	T	Inline a function at a given call site.
Label removal	T	Recognize structured gotos and replaced with continue/break.
Live variable	A	Traditional live variable analysis. Determines which variables are live at each point in the program.
Loop inversion	T	Change while loops to an if followed by a do-while.
Loop unroll	T	Traditional loop unrolling for loops with constant bounds.
Read-Only Data	A	Determine which scalar and array data is only used in a use context.
Reassociation	T	Puts expressions into sum-of-product, left associative, constants-in-front form.
Statistics	A	Counts nodes and variables in the MIRV tree.
Strength reduction	T	Convert complex operations into simple ones (multiply to shift, divide to shift, mod to mask).
Web splitting	T	Splits a given variable into several variables, one for each web.

Table 7.4: The MIRV analysis and transformation filters.

```
for                    for                    for
(i=0;i<DIM_X;i++) {    (i=0;i<DIM_X;i++) {    (i=0;i<DIM_X;i++) {
  B[i] = 0;              B[i] = 0;              t1 = &B + i*4;
                         t1 = &B + i*4;         t2 = 0;
  for                    for                    for
(j=0;j<DIM_Y;j++)      (j=0;j<DIM_Y;j++)      (j=0;j<DIM_Y;j++)
    B[i] += A[i][j];       *t1 += A[i][j];        t2 += A[i][j];

                                                *t1 = t2;
```

Figure 7.13: Example of promotion of an indirect memory reference.
After optimization, the invariant address computation is moved out of the inner loop, leaving a dereference of t1. This constitutes a load and store operation. Note that the address arithmetic is now explicit (no C scaling). After register promotion, the dereference is removed from the loop [Brig92].

the number of branches in the loop, which turns while loops into do-while loops.

4. The variable constituting the base address must be loop invariant and there must be no other definitions of it in the loop.

Step 4 in the above algorithm depends on the alias analysis used in the compiler. MIRV does flow-insensitive, intra-procedural alias analysis. Our alias analysis is based on the presentation in Muchnick’s book, where he also notes that flow-insensitivity is not usually problematic [Much97]. The lack of inter-procedural alias analysis implies that pointers passed in function calls result in the conservative assumption that anything reachable through the pointer is both used and defined in the callee. To mitigate the effects of function calls, the compiler performs a simplistic side-effect analysis [Bann79] For each function, the compiler records which global objects may be used and defined in the function. Pointer arguments are assumed to point to any global object of compatible type. We have observed that this process opens up many more chances for promotion than would otherwise be possible.

The MIRV promoter can also promote dereferences of invariant pointers. For example, the loop in Figure 7.13 has an array reference with a loop-invariant index. The array element can be promoted to a register as shown in the figure.

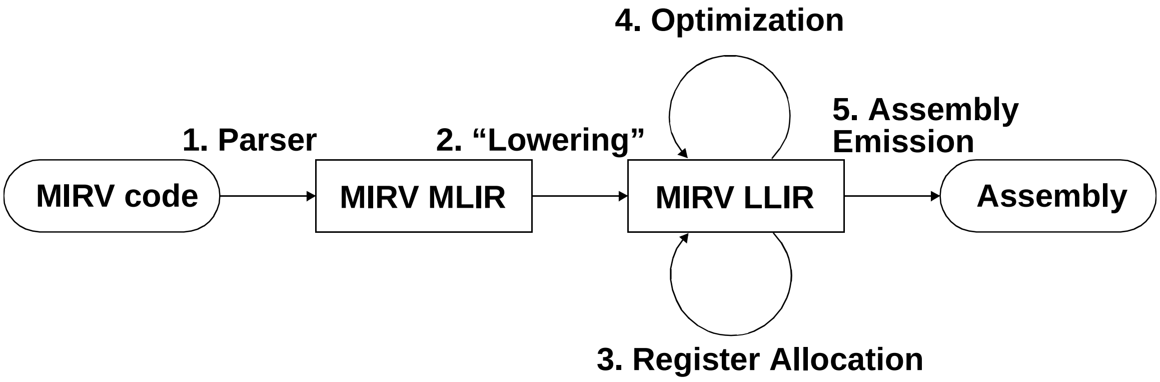


Figure 7.14: Operation of the MIRV backend

7.8. The MIRV Backend and Low-Level Intermediate Form

The MIRV backend is the portion of the compiler responsible for translating MIRV IR into assembly code for a specific target architecture. Currently, the compiler targets the Intel IA32, SimpleScalar/PISA, and ARM instruction set architectures. It accomplishes the translation task in five major phases. Each function in the input MIRV is processed separately by the backend according to the following phase order (the backend does only simple inter-procedural optimizations):

1. Convert MIRV into the Medium Level IR (MLIR). The MLIR is a quad representation of the function which assumes an infinite number of symbolic registers. Each quad contains an operator, a destination, and two source operands[3]. This MLIR is structured around a basic block representation. The control flow instructions refer to labels which are names for basic blocks. The MIRV compiler backend IR operations are listed in Table 7.5.
2. Simplify the MLIR. This task, called *lowering*, simplifies each quad to the point that it can be executed directly on the target architecture. The resulting IR is called the Low Level IR (LLIR). An important part of this step is the handling of machine-dependencies such as instructions which require specific registers. Instructions are inserted into the IR which copy the symbolic register to or from a machine register. Later, the coalescing phase of the register allocator

[3] There are some exceptions. The MLIR instruction called "block copy" takes three sources. Multiply-accumulate is another example that takes three sources.

Operator Class	Operator
Miscellaneous	literal, lit, ulit, blit, comment, info, nop
Data reference	init, skip, intDataBegin, intData, intDataEnd, roDataBegin, floatDataBegin, floatData, floatDataEnd, stringDataBegin, stringData, stringDataEnd
Function	funcBegin, funcEnd, localStack, funcRet, funcArg, ffuncArg, funcCall, funcICall
Assignment	assign, blkcpy, fassign, ftoiassign
Arithmetic	add, incr, fadd, sub, decr, fsub, neg, fneg, mul, fmul, divide, fdivide, mod, ipow, fpow, isqrt, fsqrt, ftrunc
Cast/Conversion	cast, fcast, itofcast, ftoicast
Bitwise	bitAnd, bitOr, bitXor, bitRol, bitRor, bitShll, bitShrl, bitShla, bitShra, bitComp
Short-circuiting	condAnd, condOr, condNot
Compare-and-set	setEq, setLe, setLt, setNe, setGe, setGt, fsetEq, fsetLe, fsetLt, fsetNe, fsetGe, fsetGt
Compare-and-branch	cmpEq, cmpbLe, cmpbLt, cmpbNe, cmpbGe, cmpbGt, fcmpEq, fcmpbLe, fcmpbLt, fcmpbNe, fcmpbGe, fcmpbGt
Jump	jump, jumpi
Switch Tables	switchInst, switchTable
Memory	addrof, deref

Table 7.5: The MIRV low-level IR operators.

attempts to remove as many of these copies as possible by directly allocating the symbolic register into the particular machine register. If no such allocation can be made, the copy remains in the code. In this way, sources and destinations of odd instructions on the x86 such as block copy (esi/edi), divide and modulo (eax/edx), shift (ecx) and return values (eax) are allocated to the proper registers.

3. Register Allocation. The infinite symbolic register set is mapped into the limited register set of the target architecture. This will be described in detail in Section 7.9.
4. Optimization and instruction scheduling. Traditional and global optimizations are performed and instructions are reordered in each basic block to increase overlap in instruction execution. The optimization phase order is shown in Table 7.1.

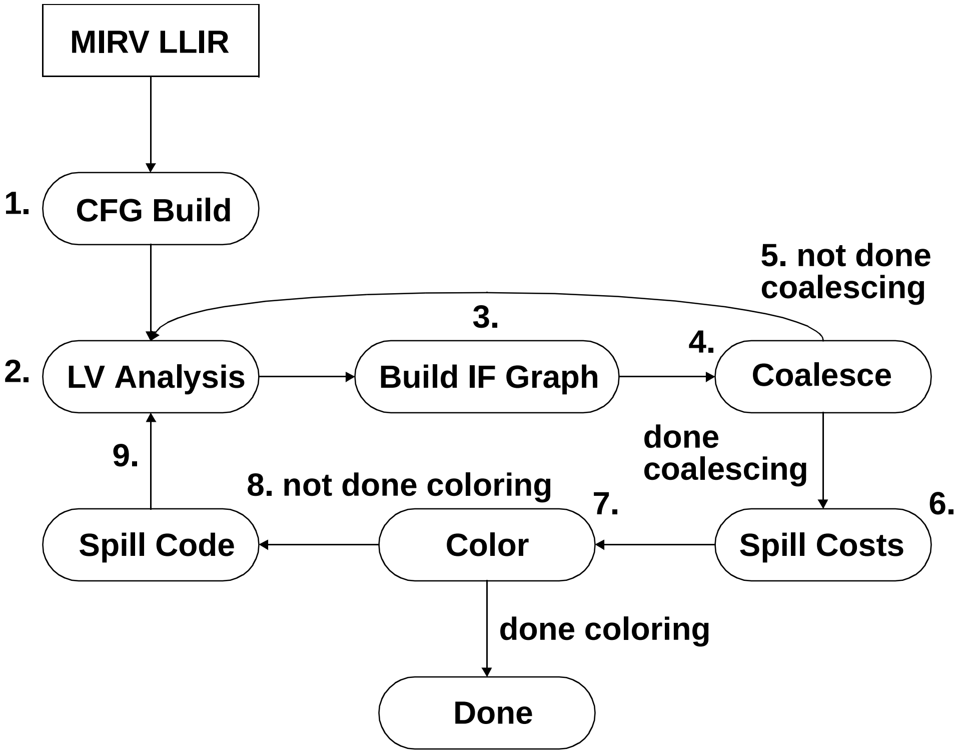


Figure 7.15: Operation of the MIRV register allocator.

5. Assembly Emission. Print out the final assembly code.

More sophisticated phase orders have been discussed in the literature; these iteratively run scheduling and register allocation or combine scheduling and register allocation into one pass but MIRV does not implement them.

The backend MIRV parser implements a syntax-directed translation of the high-level IR into a basic-block structured low-level IR. It does not use techniques like tree matching used in GCC or LCC [GCC, LCC].

7.9. Register Allocation and Spilling

The MIRV backend's register allocator performs the following steps. Explanations are simplified for the sake of brevity and we assume the reader has some familiarity with register allocation terminology [Chai81, Chai82, Brig92].

1. Build the control flow graph (predecessor and successor information)
2. Perform variable definition/use analysis and iterative live variable analysis

3. Build the interference graph. The MIRV register allocator is modeled after a standard Chaitin/Briggs register allocator [Chai81, Chai82, Brig92]. The allocator assigns the symbolic registers to machine registers. Since in general there are more symbolic registers than machine registers, the allocator algorithm performs a packing algorithm which in this case is cast as a graph coloring problem. The nodes of the graph represent machine and symbolic registers and the edges represent interferences between the nodes. The interferences indicate when two values cannot be assigned to the same register. This is determined by examining the live variable analysis results and determining what variables are simultaneously live.
4. Coalesce nodes of interference graph. Coalescing has several important features: it removes unnecessary copy instructions; it precolors certain symbolic registers; it allows easy handling of idiosyncratic register assignments for special instructions and for the calling convention; and it allows the compiler to seamlessly handle two-operand architectures such as the x86. Coalescing will be described in more detail in the next subsection.
5. Repeat steps 2 through 4 until no more coalescing can be performed.
6. Compute spill cost of each node. Variables which are used more frequently (as determined by a static count) are given higher spill costs. The spill cost determines which variables are selected first for spilling during the insertion of spill code.
7. Attempt to color graph. As in [Brig92] our allocator speculatively assumes that all nodes can be colored and only when a node is proved not to be colorable is it spilled. Assign a color to every node that can be safely colored.
8. If the graph is not colorable, insert spill code. All registers which were not assigned a color by step 7 are spilled. Spilling will be described in a later subsection.
9. Repeat steps 2 through 8 until the interference graph is colorable.

7.9.1. Coalescing

Register coalescing combines nodes of the interference graph before coloring is attempted. It attempts to transform copy instructions of the form assign sX -> sY and allocate symbolic registers sX and sY to the same register. It can do this if sX and sY do not interfere (their live ranges do not overlap). If so, the assignment can be removed, and all references to sX are changed to refer to sY. We say that sX has been coalesced with sY and the result register is sY. This results in the elimination of the sX node from the interference graph which simplifies the later graph coloring step. This optimization is really global copy propagation, since it removes a copy instruction and propagates the result register to all other use and definition sites. It is very effective since it runs late in compilation.

Which of sX or sY we choose to keep as the result register is only a matter of implementation, except when one of the sX or sY is a machine register. Then the coalescer must keep the machine register because a machine register cannot be removed from the interference graph. This is useful, as was mentioned before, because it allows the earlier IR lowering phase of the backend to insert fixup code which contains machine register specifiers. This fixup code is executed without regard for how many copy instructions are inserted or where they are inserted. The fixup code simply ensures that if a specific register assignment is required by the architecture, a copy instruction is inserted whose source is a symbolic register and whose destination is a machine register (or vice versa). Coalescing also assists in handling two-operand architectures such as the Intel IA32, as will be explained below.

The example in Figure 7.16 demonstrates why these features of coalescing are useful. In the C code, a left shift operation is requested. The IR equivalent is shown in (b). On the IA32 architecture, the shift amount must be placed into the `%ecx` machine register, so the lowering phase inserts a copy of sB into `%ecx` (c). It also inserts a copy of sA to sD because the IA32 only allows two operand instructions (a = b + c must be implemented as two instructions: a = b and a += c)[4]. In the example, two coalescing operations occur between steps (c) and (d). The first coalescing operation occurs because sB is copied into %ecx and those two registers do not interfere (b). So sB is coalesced into %ecx, effec-

[4] This explanation is simplified, as will be shown later in this chapter.

C Code	IR	Lowered IR	Machine Code
a = 1;	assign 1 -> sA	assign 1 -> sA	movl 1,%eax
b = 13;	assign 13 -> sB	assign 13 -> sB	
...	...	...	...
d = a << b;	sD <- sA << sB	assign sB -> %ecx	movl 13,%ecx
		assign sA -> sD	
<a and d don’t interfere after this>	<sB doesn’t interfere with %ecx>	sD <<= %ecx	shll %ecx,%eax
(a)	(b)	(c)	(d)

Figure 7.16: An example demonstrating the utility of register coalescing.

C Code	IR	Two-operand Lowered IR

a = b + c	sA = sB + sC	sA = sB
		sA = sA + sC

(a)	(b)	(c)

Figure 7.17: Coalescing for a two-operand architecture.

tively pre-allocating sB before the graph coloring even runs. The second coalescing operation occurs because sA and sD are known not to interfere (from (a)). The lowering phase can insert as many copies as it needs to put the code into a form that is acceptable to the target architecture, and the register coalescer removes (propagates) as many of those copies as it can by pre-allocating symbolic registers into other symbolic or machine registers.

The pre-allocation feature of coalescing is particularly useful to handle specific register assignments of the machine architecture. But the IA32 architecture also requires that operate instructions such as addition take only two operands – one source and a destination which doubles as a source. In other words, a = b + c is not valid for the IA32, but a = a + b is valid. The naive solution to this problem is shown in Figure 7.17. Here, the three-operate format a = b + c has to be broken down into two simpler instructions – a = b and a = a + c. This is the technique used in [Brig92]. We call this process “inserting the two-operand fixup instructions.”

There are two problems with the solution shown in Figure 7.17. As far as we know, neither problem has been described in the literature, perhaps because experiments were always run on a RISC (3-operand) architecture and not verified on the (2-operand) IA32. The first is shown in Figure 7.18. In this case, a = b - a, the simple transformation described above overwrites the value of a, which is needed as the second source of the computation. In general, this kind of sequence requires three instructions and a temporary variable to be correct, as shown in Figure 7.18(d)[5].

The second problem with the solution used in Figure 7.17 is more subtle because it generates suboptimal (but correctly functioning) code. The two-operand simplification was performed in the IR lowering phase. However, the best place to do the two-operand modification is in the coalescer and not in the IR lowering phase. This is because the coalescer has information regarding interference of registers. This is convenient because it allows the selective insertion of two-operand fixup instructions. In MIRV's implementation of this more sophisticated approach to coalescing, the backend uses the following steps to determine if it can coalesce the operands of an instruction:

1. For a copy instruction of the form a = b, coalesce a and b if possible. The remainder of the algorithm is checking for two-operand optimizations.
2. If the machine requires two operand instructions, check the destination and first source operand. If they can be coalesced, do so. In this case, we avoid inserting any fixup code for the two-operand machine.
3. Again, for a two-operand instructions, if step 2 did not work, examine those instructions which are commutative. Check dest and source2. If they can be coalesced, do so and swap source1 and source2.

C Code	IR	Incorrect Two-operand Lowered IR	Correct Two-operand Lowered IR

a = b - a	sA = sB - sA	sA = sB
		sA = sA - sA
			sT = sB
			sT = sT - sA
			sA = sT

(a)	(b)	(c)	(d)

Figure 7.18: Failure of the simple-minded coalescing algorithm.
For a two-operand architecture, the algorithm does not work. The value of sA has been overwritten in (c). The correct code is shown in (d).

[5] There are some cases which could be specialized by using negate or some other clever sequence of instructions; our compiler chooses the straightforward approach shown in the figure.

C Code	IR	Two-operand fixup in lowering stage	Two-operand fixup in coalesce stage

a = b + c
<a and c do
not
interfere, a
and b do>
	sA = sB + sC	sA = sB
		sA = sA + sC
			sC = sB + sC
			<sA has been
			allocated to
			sC’s
			register>

(a)	(b)	(c)	(d)

Figure 7.19: Early insertion of two-operand fixup code.
Knowing the register interference information allows more effective coalescing. Since variables a and b interfere, the early coalescing algorithm gives up and inserts a copy instruction, as shown in (c). However, since variables a and c do not interfere, a can take up residence in c’s register after the addition operation. The final step in (d) is to swap the sB and sC operands of the addition (not shown).

4. If both steps 2 and 3 did not coalesce the operands, then no coalescing can be performed, either because the operator is not commutative, or both operands are simultaneously live with the destination. Only in this case do we need to insert the two-operand fixup instructions.

If two-operand fixup instructions are inserted too early (in the lowering phase), the opportunity to swap operands later and coalesce them is lost forever. This produces less than optimal code. This is demonstrated in Figure 7.19. Since the coalescer knows that a and c are not simultaneously live, it can swap their positions in the add instruction and allocate them to the same register and avoid any two-operand fixup instructions. The IR lowering phase cannot know this since it has not computed liveness information, and so must insert the two-operand fixup. Since a and b do interfere, no coalescing is possible in Figure 7.19(c).

7.9.2. Spilling

During graph coloring, a node may be encountered which we cannot guarantee to receive a color because it has more neighbors than colors (registers) available. In this case, we may need to generate spill code. There are three major phases to the insertion of spill code.

1. Before coloring begins, the spill cost of each node in the graph is computed. Machine registers receive an infinite spill cost (machine registers cannot be spilled in our algorithm). Symbolic registers receive a cost which is based on their static usage count. This count is called the “frequency count” and is weighted by loop nesting level so that uses in loops receive higher usage count. Compiler-generated temporaries are given a slightly higher spill cost because they typically have very short live ranges and spilling them would hurt performance and not significantly impact the colorability of the interference graph.
2. During coloring, if we find that we cannot remove any nodes from the graph which have degree less than or equal to the number of colors (registers) available, we select a spill candidate. The symbolic register selected for spilling has the lowest spillCost / degree ratio. This is a common heuristic for selecting a spill candidate [Brig92] which tries to select a spill candidate with low spill cost (to reduce impact on execution time) and one which will remove a lot of edges from the graph (to make the remaining nodes easier to color). Once the spill candidate is selected, it is speculatively assumed to be colorable and the coloring algorithm continues. Only after color assignment finishes and we have determined that in fact no color was assigned to a spill candidate do we actually insert spill code for it. The non-speculative version of this algorithm would, upon selecting a spill candidate, immediately spill it.
3. After we determine that there were some nodes that were not colored, we insert spill code for those nodes. This consists of walking the list of registers, and for each one that did not receive a color, we insert a load before each use of that symbolic register and a store after each definition of it. A later peephole pass can eliminate redundant loads and stores. The loads and stores effectively chop up the live range of the variable into many small pieces. When the next interference graph is built, there will be fewer edges because the spilled symbolic register has very short lifetimes (much like a compiler-generated temporary).

Live range splitting and rematerialization are not present in the current version of the register allocator. These optimizations reduce the cost of spill code [Brig92].

7.10. Comparisons to other compilers

Currently the only compiler which has been compared to MIRV is the SimpleScalar/PISA port of the gcc-2.7.2.3 compiler. The details are reported in [Post00a]; they include comparisons of execution time, cache performance, instruction mix, and a number of other characteristics.

Future work will compare MIRV to other compilers in terms of performance and regression test success. Primarily, this includes the newest version of the gcc compiler, which has better optimizations than the version we compared against above [GCC]. Other compilers which could be tested include LCC [LCC], the SGI Pro64 Compiler [SGI], Greenhills [GHS], Impact/Trimaran [Impact, Trimaran], and the National Compiler Infrastructure Initiative/SUIF [SUIF], and the DEC/GEM Compiler [Blic92]. These could easily be tested on our regression sources. A direct comparison of performance would require retargeting either MIRV or these other compilers to a common target ISA.

7.11. MIRV Example

This section briefly describes the progression of a compilation by presenting an example which shows how the source code proceeds through the compiler to assembly code.

The source code we will be using for this example (Figure 7.20) is a simple C program which does some arithmetic and prints out a few variables.

The MIRV code for our example is shown in Figure 7.21, after the MIRV frontend has processed it. The initial symbol table information and variable declarations are not shown to conserve space. The explanation of MIRV in Section 7.5 covers most of the elements of this of code.

The backend reads this code and produces a (relatively high-level) low-level IR representation of it. This is shown in Figure 7.22. The operators of this language were previously described in Section 7.8. The `localStack` keyword is a marker for the backend

```
extern int printf(char *,...);

int main(void)
{
  int i, j, k, l;
  i = 0;
  j = 15;
  k = j;
  l = -k;
  i = j + 1;
  printf("i=%d, j=%d k=%d l=%d\n", i, j, k, l);
  return i;
}
```

Figure 7.20: Example C source code.

```
 fdef unid main {
   assign vref unid main._5_7_i  cref unid sint32_0
   assign vref unid main._5_10_j cref unid sint32_15
   assign vref unid main._5_13_k vref unid main._5_10_j
   assign vref unid main._5_16_l neg vref unid main._5_13_k
   assign vref unid main._5_7_i
      + vref unid main._5_10_j cref unid sint32_1
   fcall unid printf {
     addrOf unid sint8p
       cref unid __mirv_pack.m1.30 # i=%d, j=%d k=%d
l=%d\n\0
     vref unid main._5_7_i
     vref unid main._5_10_j
     vref unid main._5_13_k
     vref unid main._5_16_l
   }
 assign vref unid main.__return_value vref unid main._5_7_i
 return
}
```

Figure 7.21: MIRV code for the example.

to insert function prologue code; the `funcret` keyword performs a similar function for the function epilogue code.

Figure 7.23 shows the low-level IR after the backend simplifies the original low-level IR. Register assignment has been performed and some spilling code has been intro-

```
global function main {
     localStack
.L3:
     assign %53,$0
     assign %54,$15
     assign %55,%54
     neg %56,%55
     add %53,%54,$1
     funcCall printf
       &$$__mirv_pack.m1.30
       %53
       %54
       %55
       %56
     assign %2,%53
     funcret
```

Figure 7.22: Low-level IR code for the example.

duced. The addition operation of the earlier code has been changed to an increment (`incr`). Since the target being considered in this example is SimpleScalar/PISA, a target-specific function call to `__main` has been inserted. This call sets up the runtime environment. The `funcret` has been replaced with an explicit jump to a label. Finally, the function call instruction has been broken into several `funcArg` instructions and the original `funcCall` has become simpler. The arguments that were listed under the `funcCall` opcode previously have been changed to individual instructions that set up the arguments. These are operationally the same as assignments but are marked with a special opcode so it is easier to determine the source of the operations.

The final step of the compilation is shown in Figure 7.24, where the SimpleScalar/ PISA assembly code is shown. Here the full detail of the machine is exposed. A stack frame is declared; this is used only for debugging purposes. The return address ($31) and the frame pointer ($fp) are stored to the stack. The stack frame is set up with by copying the stack pointer into the frame pointer rand subtracting 64 (the size of the frame) from the stack pointer. Then register 23 ($23) is stored to the stack below the frame pointer. This is necessary because $23 is a callee saved register and the callee (in this case, main), is using that register for a temporary. The initial call to `__main` is performed, and then the body of

```
global function main {
     localStack
     assign $-12+(%30),%23  # spill
     funcCall __main
.L3:
     assign %23,%0
     assign %6,$15
     assign %7,%6
     neg %2,%7
     incr %23,%6
     funcArg %4,&$$__mirv_pack.m1.30
     funcArg %5,%23
     funcArg $16+(%29),%2
     funcCall printf
     assign %2,%23
     jump L4
L4:
```

Figure 7.23: Low-level IR code for the example, after further lowering.

the code begins. The low-level IR opcodes have been transformed to their appropriate SimpleScalar/PISA (MIPS) counterparts. Notice the attention to unsigned arithmetic in the `subu` and `addu` opcodes. This is necessary in C because the PISA/MIPS instruction set architecture defines that an signed addition or subtract will have an undefined result in the case of an overflow; C semantics say nothing of overflow, so all arithmetic operations are emitted by the backend as if there is no signedness. The address of the `printf` format string is then loaded into the first parameter register ($4), and the result of the add operation is put into the second parameter register ($5). The other parameters were calculated directly into their parameter registers ($6 and $7). This is a result of the register allocator and coalescing algorithm described earlier. The value of variable `i` (in $23) placed into $2 as the return value, the function restores register $23, restores the stack pointer, restores the return address register, restores the frame pointer, and finally returns.

7.12. Test and Debug Environment

The success of any large software project depends to a large degree on the testing environment that is built to ensure correct operation of the software. When an incorrect

```
main:
     .frame $fp,64,$31
     sw   $31,-4($sp)
     sw   $fp,-8($sp)
     move $fp,$sp
     subu $sp,$sp,64
     sw $23,-12($fp)
     jal __main
L3:
     move $23,$0
     li $6,15
     move $7,$6
     subu $2,$0,$7
     addu $23,$6,1
     la $4,$__mirv_pack.m1.30
     move $5,$23
     sw $2,16($sp)
     jal printf
     move $2,$23
$L4:
     sw $23,-12($fp)
$Lmain_ret:
     move $sp, $fp
     lw $31,-4($sp)
     lw $fp,-8($sp)
     j $31
```

Figure 7.24: SimpleScalar/PISA assembly code for the example (unoptimized).

operation is discovered, good debug tools are helpful to speed the bug discovery and correction process. This section describes the test and debug tools developed for MIRV.

7.12.1. Regression testing

The simplest form of testing for the compiler was also the first one developed. Every time a bug is discovered, a test case which exposes the bug is manually generated. Before committing any new changes to the source tree, all previous test cases are run to ensure that no new bugs have been introduced. Presently, there are 472 test cases and each of these is run at three optimization levels during regression testing. These test cases actually exposed two bugs in the version of SimpleScalar/PISA gcc that we use (2.7.2.3 with GNU binutils 2.9.5+).

All the SPECint95 and 2000 benchmarks that MIRV compiles (presently 19) are also run before each source tree modification is allowed to be put into the source repository. Each benchmark is run at the same three optimization levels as in basic regression testing.

Finally, tests can be run on the SPECint95 benchmarks over seven data sets and three optimization levels. These are used to periodically evaluate the compiler's performance. Statistics are tracked at the function level to make it easier to find performance regressions.

7.12.2. The bughunt Tool

When a bug is discovered in a multi-object-file program, the bughunt tool can help to isolate the bug. It compiles a known-good version of the program using either gcc or MIRV with -O0 optimization. This process produces a set of N known-good object files. It then compiles a version of the binary with the optimization flags in question. One or more of the object files produced during this compile is bad, but all of them are questionable since it is not known which is bad. Each of these questionable object files is linked with the other known-good object files and tested. In time linear with the number of object modules in the program, bughunt finds out which object modules have been compiled incorrectly.

There are several considerations that we found to be necessary during the development of this tool. First, some object modules can cause the program under test to go into an infinite loop. This necessitates a timer function in bughunt which times out after a certain amount of time passes. This amount of time is determined by multiplying the time it takes for a known-good binary to be tested by some factor (in our case, 3).

Occasionally we have discovered incompatibilities between the code produced by the known-good compile and code produced by the questionable compile. Sometimes, for example, the PISA version of gcc produces code which does not conform to the UNIX System V Application Binary Interface (ABI). This occurs in the areas of structure and bitfield layout, for example. Such incompatibilities lead to the report of false bugs.

7.12.3. The cleaver and machete Tool

Once a bug is found in an object module, it may still be an onerous task to isolate the bug to the function and line in the source code where it occurs. The cleaver and machete tools help in this regard. Cleaver takes a MIRV file and chops it into a given number of pieces (or one piece per function). Machete is similar but chops up an assembly file into pieces. We will only consider the cleaver tool from this point on.

Given a MIRV file with 10 functions in it and some global variables, cleaver chops the MIRV file into 10 MIRV files - one per function - plus another MIRV file for global variable definitions, and one more for declarations that must be included into each of the other MIRV files. This is conveniently done because of the high-level structure of the MIRV code. These MIRV files can then be compiled to object modules and used with the bughunt program described in the previous section. Cleaver and bughunt can be used to isolate the bug to the very function that is causing the error.

7.12.4. The brutal tool

The brutal tool is one which attempts to find new bugs in optimization filter sequences. Given a set of test source files and a list of optimizations, it tries random selections and permutations of the optimizations on a random selection of the test files. It will also focus on just a single test file, if desired. This is useful for finding bugs which are only exposed by a certain sequence of optimizations. For example, one bug was found which required a pass of reassociation, followed by register promotion, followed by low-level common subexpression elimination. Brutal can be run for a given number of iterations or forever. The failing compiler commands are sent to a database for future regression testing.

7.12.5. The lmrvOptimizeAndRun Tool

This tool simply takes a linked MIRV file and runs it through the compiler with a given set of flags, runs the program, and determines if the output was correct. This is useful for debugging large programs as will be described in the next section.

7.12.6. The cmdshrinker Tool

The cmdshrinker tool takes as an argument a Unix command string, in our case, a compile command. This command has several flags in it to specify optimizations. The cmdshrinker tool attempts to remove as many of those flags from the command line while retaining a bug that was exposed by the original command string.

For example, if the failing command is mirvcc -fcleaner -fregPromote -fCSE -fLICodeMotion test.c, then command shrinker could determine, in time linear in the number of filters specified, that mirvcc -fCSE -fLICodeMotion test.c is the smallest command string that fails. If any of the remaining flags were removed, the bug would disappear. This effectively narrows in on the offending filter. One filter from that command could be removed and then the MIRV or source code could be compared against the failing version. the differences can show where the bug occurs.

7.12.7. Transformation Limit Tool

Once a bug is narrowed down to a particular optimization filter (say with the cmdshrinker tool), the transformation limit tool allows the programmer to "dial in" the exact transformation that caused the error. For example, if the register promotion filter has a bug, we can run mirvcc -fregPromote=--maxPromotes=N. Knowing 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. It can then be determined why the promoter thought a transformation was legal when in fact it was not. This technique can be used on many optimization filters: loop invariant code motion, register promotion, common-subexpression elimination, loop unrolling. In general, it can be applied to any transformation filter where the transformations can be applied independently of one another.

7.12.8. Miscellaneous Test Generator Tools

During the course of building the compiler we also found it helpful to have large or complex test cases generated by scripts. One such script generates loops to use for unrolling or other loop optimizations. Another such script generates a program which call functions with all combinations of parameter types. This is useful for testing whether the compiler implements the ABI correctly, that is, by passing the correct parameters in registers or on the stack.

7.12.9. The mirvbot Tool

The MIRV robot tool automates overnight testing of the compiler for correctness and performance. Each night a number of the SPECint95 benchmarks are compiled and checked to determine that they run correctly. Statistical printouts are also gathered which show the performance of the compiled binary in a number of areas - number of cycles, dynamic instructions, memory references, cache misses, etc. These can be examined daily and over time to see how modifications to the compiler have affected performance.

7.12.10. The ssprofcomp tool

This tool gathers profile information from any number of SimpleScalar simulator variables and attaches them to the functions in the program that are responsible for the changes in the simulator statistics. For example, it shows if a particular function is causing most of the cache misses or if a function is responsible for all of the memory accesses. This directs the optimization tuning effort to look at the most important functions.

7.12.11. Select Optimization Subset (SOS)

One problem that rises during debugging very large MIRV files is the large number of functions to optimize. To wait for optimizations to happen on all the functions is wasteful because only the one function which has the bug in it matters. MIRV has the ability to run filters on a single function or all functions up to and including some function (in source order). In addition, MIRV can run a filter subset on all functions before the target (last) function in the function subset. This is useful for performing tests of whole program optimization. For example, several bugs were found in def-use analysis when it carried information from one function to the next in our simple interprocedural analyses. Thus we could run -fdefUse on those functions up to the one we discover the bug in, then fully optimize that function, then skip optimizations on all the rest of the module. This relies, of course, on the fact that the non-local (e.g. global and parameter point-to) def-use information which summarizes a function does not change with optimization.

7.12.12. Generic findbug

With all of these 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. Findbug is one such tool.

At the simplest level, findbug is composed of two phases: a binary search phase to find the function in the source program that is badly compiled and a command shrinker phase to reduce the number of optimizations that are run on the file to the minimal number needed to expose the bug.

In the first phase, findbug makes use of the ability of MIRV to restrict filter operations on a subset of source functions, as described in the previous section. The binary search phase of findbug uses these options in a variety of combinations to home in on the source function which is being badly compiled. To do this it uses the “binsearch” library. First, binsearch performs an lmrvOptimizeAndRun pass on the source program with the specified options passed to the compiler. All functions in the source program are transformed. lmrvOptimizeAndRun produces a return code which is saved off to the side. The binsearch then runs lmrvOptimizeAndRun as before, but adds arguments to compile every function up to and including the halfway point through the program. If the return code is the same as before, the search space is restricted to the lower half of the program, meaning that the errant compilation happened on a function in the first half of the program’s list of functions. If it is not the same, the higher half of the program is searched. The binsearch phase progresses as a standard binary search and the number of the offending function is returned as the final result. Upon receiving the suspect function number from the binsearch phase, findbug invokes cmdshrinker with the same arguments as binsearch, but adding flags to restrict transformation only to the function returned by binsearch. This greatly speeds the cmdshrinker process. In addition, flags are passed to run alias and reaching definition analyses so that interprocedural dataflow information can be used in the transformation process (if such action is specified in the mirvcc command line passed to findbug). cmdshrinker reduces the filter list and prints the final offending command so the user can re-run and examine the (incorrect) actions of the compiler.

Generally, we have found that once a command line is returned by findbug, removing a filter from the command-line, running it and comparing the (correct) output with the (incorrect) output obtained by running the findbug result command produces such a small number of differences that the cause of the error is immediately obvious. In the case of a large number of differences, the transformation limit tool can be used to narrow the scope.

7.12.13. Future Tools

There is also a need to write a “testshrinker” tool which will take a failing program and shrink it to the minimal source file which still produces the bug.

7.13. Profiling Support in MIRV

MIRV presently has support for several kinds of profiling. The most basic is the block profiling filter. Others include the value profiler and the time profiler. These will be discussed in turn.

7.13.1. Block Profile Filter

The block profiling filter reads a linked MIRV program and produces one which, when run, will produce a data file containing a counter for each block statement in the program. The value of the counters equals the number of times that the block statement was reached during the execution of the program.

Block profiling is actually done with a combination of two filters. The first instruments the program by adding the code to increment the counter values at each block statement. An array of counters is dynamically allocated before the program run begins and the array is dumped to disk immediately at the exit of the program.

After running the program on a training data set, the counter values are then *back annotated* into the original program source as attributes on the MIRV block statements. These attributes can then be used by filters for a number of analyses or optimizations. In our case, we use the counts to determine how many times a variable is referenced in the program. For example, if a variable is referenced in two blocks whose counts are 10 and 15, then the variable is given a frequency count of 25. This dynamic frequency count of the variable can be used to determine the importance of the variable in terms of register allocation.

The count data produced by the block profiler is not completely accurate because block statements do not always correspond to basic blocks in the program's object code. Thus, there are a few cases such as unstructured code and early exits from loops where the block counting will not produce completely accurate results.

Block profile data can be used to direct optimizations. At this point, it is used to direct the register promotion optimization to avoid promoting in regions that are not profitable.

7.13.2. Value Profile Filter

The value profile filter tracks the values of function arguments and keeps track of the most common sets of arguments passed to each function. This is useful for performing function specialization where functions which take a certain set of arguments often are cloned and optimized especially for those parameters [Green98].

7.13.3. Time Profile Filter

The time profile filter back annotates execution times for each function and cumulative times for a function including all of its children, onto the MIRV tree. An instrumented version of the MIRV code is run, and the timing data dumped into a gprof-format output file [GPROF]. The compiler can read in the MIRV file and the timing data and back annotate the appropriate times into the MIRV tree.

Timing data is useful for optimizations such as inlining and specialization, where optimizations should only be focused on heavily used functions.

7.14. Approach to High-Level Dataflow Analysis

The design and implementation of the MIRV dataflow analysis infrastructure is described in early work done by Flautner, et. al. [Flau97, Flau98b, Flau98c], all available at the MIRV website [MIRV].

7.15. MIRV Simulator

Another program that operates on linked MIRV programs is the MIRV simulator [Flau98a]. This program simulates a MIRV program and can annotate the MIRV IR with information about path frequency, branch prediction accuracy, data reference patterns, and a host of other information. It implements system calls with stubs.

7.16. Software Engineering Considerations

During the course of developing the MIRV compiler, many software engineering concerns have been raised. This section gives the general software design principles that we have learned during the course of this project.

1. Simulation and compiler code should be designed to fail fast. Techniques such as using assertions and panics liberally are one way to do this. Another way is to use the proper simulator in SimpleScalar. For example, the `sim-fast` simulator is an unsafe simulator that may produce correct output for an incorrect program. We ran into several instances where this occurred; simply switching over to the `sim-safe` simulator caused the bugs to be manifested earlier in the program (or to be manifested in the first place).
2. Make liberal use of other’s code. Library code such as STL and well-known design patterns have saved a lot of time. The visitor, singleton/manager, and factory patterns have been especially useful in this project.
3. Be paranoid about your own code. Assume that it is wrong and build in checks to make sure that it is actually working the way you want. While this takes additional effort up front, it saves a lot of time later and gives much more confidence in the results.

4. In lieu of garbage collection, implement strict resource ownership and ownership counting mechanisms in class objects. This exposes bugs sooner and helps to more quickly track them to their source.
5. When a manager structure is accessed to find an object, do not create one automatically for the user with default values. Assume that the programmer forgot to create the object and return an error. Numerous logical errors resulted from our default attribute creation code. In addition, memory usage became a problem at times because default objects were being created in places where they were never needed; exposing these early by returning an error code would allow these typos in the compiler code to be fixed earlier.
6. Use simple C++. This was particularly true in the earlier days when g++ could not handle the newest C++ features. Limiting the use of the language also facilitates multi-programmer development since each programmer need not be familiar with the coding style used by the most sophisticated programmer.
7. The importance of data structure design cannot be overemphasized. Design twice and code once is a rule which should be changed to "design three times and code once." Many errors and inefficiencies were found by thinking through the algorithms that were implemented in the compiler.

In addition to those general principles there were several shortcomings that we discovered with the STL library that we think relevant:

1. The size member in the list class is very inefficient. This design was required to make other list operations fast, but required that we had to code around it by keeping track of the size ourselves.
2. Part of the above problem is that iterators do not know about the container that contains them. Thus, when an element is added to the middle of a list, the list cannot be informed to the change. Thus, the list cannot keep track of the number of elements that it has and must resort to counting the list elements every time list.size() is called.
3. String iterators are different than the iterators in the rest of the C++ library. This makes coding a little difficult.

4. Lists do not provide convenient += operators (only ++). The += operator is present in the advance() routine.

7.17. MIRV as a Teaching Tool

MIRV originated in 1995 from a class project in EECS 483. It is presently used in the same course as well as the advanced compiler course EECS 583 at the University of Michigan. In EECS 483, students write a lexical analyzer for a C-language subset, then parse it and produce MIRV IR. The next assignment is to read the MIRV IR into a backend lexical analysis and parsing phase and produce code for the SimpleScalar/PISA target. The definition of the MIRV language, regression tests, and a number of tools are supplied to the student to make this task reasonable to complete in a semester. By the end of the semester, they have a working, though non-optimizing, compiler for a RISC target and a subset of C.

The next course, EECS 583 course builds on that experience. The students are given the source for the MIRV compiler, minus some functions which implement key analyses and transformations, and they are asked to reproduce those functions. Later in the course, student groups work on a compiler-related project. A number of groups have attempted to retarget the compiler to different architectures; among these are ARM and PowerPC.

7.18. Conclusions

This chapter has described the MIRV C compiler. The compiler was used as the primary tool in the research presented in this dissertation. In addition, there are a number of novel features in the compiler, such as the high-level intermediate form, regression testing and debugging environment, MIRV high-level linker, MIRV simulator, details of implementation of the register allocator, implementation of dataflow analysis using design patterns.

A number of elements of MIRV were not described here. For example, the internal operation of the compiler, how to add optimization filters, the detailed operation of each backend target, and other basic instructional items are left for the online documentation and source code comments.

Chapter 8
Conclusions

In this dissertation, we have discussed a number of closely related issues. After introductory material in Chapters 1 and 2, Chapter 3 showed that aggressive compiler optimizations require a large logical register file, in contrast to some previous work. When all of the optimizations are applied, performance increases of up to 45% were possible. Memory reference operations were reduced by 40% on average, and 60% or more in some cases.

The primary challenge with a large logical file is the access time. Therefore in Chapter 4 we proposed a new way to implement a large logical register file using a new caching technique integrated with register renaming and physical register storage in an out-of-order processor. We showed that a fast large register file improves performance by about 20% over an un-cached (slower) register file and our caching technique comes close to the performance of the fast register file.

In Chapter 5, we demonstrated that another technique to make more effective use of processor registers, register promotion, was limited because of compiler alias analysis. We proposed a new hardware structure called the SLAT and a compiler optimization to allow register promotion to be applied to a larger number of variables. Benefits of this approach were found to be largely benchmark-dependent, but it can eliminate up to 35% of loads and up to 15% of stores for some benchmarks. This benefit can be achieved in the context of separate compilation, though the whole-program optimizations presented in Chapter 3 are easier to implement.

Chapter 6 then introduced some new metrics to quantify the effect of compiler optimization on hardware complexity and analyzed the benchmarks to determine characteristics of the memory operations. The issue-width equivalent metric is used to show how the compiler optimizations compare to additional complexity in the issue logic of the processor. The results showed that the optimizations were worth 1 or 2 issue widths of performance; that is, a binary compiled with advanced optimizations and 256 registers on a 2-issue processor performed just as well as an binary compiled with -O1 and 32 registers on a 4-issue machine. Similarly, the cache-latency-equivalent metric was used to show how compiler optimizations could be used to tolerate additional latency in the L1 data cache. For some benchmarks, the L1 data cache can be removed entirely. Both of these analyses showed that the advanced optimizations themselves (not basic optimizations used in all compilers) were responsible for the performance benefits. Further analysis presented in Chapter 6 showed a categorization of memory operations in several benchmarks. The memory operations that are responsible for cache accesses and misses typically have complicated addressing computations in loops and sometimes a memory address is dependent on an immediately preceding load instruction. These are very difficult to optimize, and since they load from a large range of addresses, the compiler cannot allocate these memory locations to registers. Other research into prefetching and miss latency toleration are necessary to address these memory operations.

Chapter 7 described the MIRV compiler, the primary tool used for this research. The frontend, optimization filters, and backend were described briefly. The MIRV language and the test/debug environment used to build the compiler are described at length. The importance of regular regression testing and a small development team to the realization of the compiler were noted.

The techniques presented in this dissertation are not strictly additive. That is, the results of the different proposals here cannot necessarily be concatenated one to another. For example, some of the optimizations shown in Chapter 3 can optimize away cases that the SLAT of Chapter 5 is intended to optimize. This particular instance arises because of the whole-program optimization and analysis facility in MIRV. Other instances (for example, inlining and register windowing) are simply targeting the same program overhead in different ways and with different levels of effectiveness.

This work can be placed in a class with other work that combines compiler, instruction set architecture, and microarchitecture research. For example, Chapter 3 showed the desirability of modifying the instruction set architecture so that a large number of registers is available to the software. Chapter 4 then examined the microarchitecture necessary to implement a large register file effectively with caching. Chapter 5 suggested changes to the instruction set, compiler, and microarchitecture all at once to improve the possibilities of register allocation.

This work can be extended in a number of ways in the future. One obvious way is to attack the memory operations remaining after optimization. This can start with a study of the effects of compiler optimizations on data prefetching performance for a number of existing data prefetch algorithms. New caching and prefetching algorithms can be developed to address the unique characteristics of the types of memory operations that were shown in Chapter 6 to be problematic. Of course, sequential structures such as arrays are sometimes accessed in sequential fashion; the compiler can easily indicate this to the hardware to allow it to do next sequential line prefetching or some derivative. Other structures such as lists are sometimes accessed in a sequential fashion but they are not laid out in memory in a sequential fashion. Other sequential structures like arrays are accessed in random fashion, such as hash tables. The compiler can be used to insert instructions that inform the hardware as to the region of memory that will likely be needed by the program in the near future, since suggesting particular addresses to prefetch may be very difficult in these contexts. Thus, the layout of the structure (sequential, non-sequential) and the primary use of it (sequential, non-sequential), must be used to determine how to successfully keep the cache misses to a minimum. Conversion of structures and primary usage from non-sequential to sequential may be helpful in attacking the remaining portions of the memory problem.

Research that trades complexity and performance between the compiler and hardware is ripe with possibility. For example, trends in the 1980s suggested a shift of complexity upward in the system organization hierarchy, toward the software and compiler and away from the hardware. This shift was successful in producing high performance RISC microprocessors for a number of years and witnessed the birth of the MIPS, Alpha, SPARC, and PA-RISC architectures. However, the push toward simpler instruction sets and hardware gave way to another trend through the 1990s, namely putting more complexity back into the microarchitecture and circuit levels to extract multiple instructions per cycle from binaries compiled in existing instruction set architectures. This enabled complex instruction set machines to become the best performing desktop and workstation processors. Most recently, this trend is shifting yet again, with the introduction of new instruction sets that support execution of old ones and put complexity into both the hardware and software in order to extract maximum performance, either to help binary translation (Transmeta) or support speculative compiler optimizations (IA64). Research that moves complexity across the instruction set architecture, the interface between the compiler and the microarchitecture, will continue to flourish as new ways of writing software are introduced and as computers move away from the traditional desktop form-factor and become smaller, more power-efficient, and take on new functionality.

Bibliography

[AD96] Analog Devices. ADSP-2106x SHARC^{TM} User's Manual. Second Edition (7/96). Analog Devices, Inc. 1997.

[Aho86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers--Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, USA, 1986.

[Ahuj95] Pritpal S. Ahuja, Douglas W. Clark and Anne Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. Proc. 28th Intl. Symp. Microarchitecture, pp. 36-45, Nov, 1995.

[Augu98] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran and W. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. Proc. 25th Intl. Symp. Computer Architecture, pp. 227-237, June, 1998.

[Ayers97] Andrew Ayers, Richard Schooler and Robert Gottlieb. Aggressive Inlining. Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97), pp. 134-145, June, 1997.

[Bala99] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent Dynamic Optimization: The Design and Implementation of Dynamo. HP Laboratories, Cambridge, MA. Technical Report HPL-1999-78, June 1999.

[Band87] S. Bandyopadhyay, V. S. Begwani and R. B. Murray. Compiling for the CRISP Microprocessor. Proc. 1987 Spring COMPCON, pp. 265-286, Feb, 1987.

[Bann79] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. Proc. Sixth POPL, Jan. 1979.

[Barr98] Luiz Andre Barroso, Kourosh Gharachorloo and Edouard Bugnion. Memory System Characterization of Commercial Workloads. Proc. 25th Intl. Symp. Computer Architecture, pp. 3-14, June, 1998.

[Beni93] Manuel E. Benitez and Jack W. Davidson. Register Deprivation Measurements. Department of Computer Science, University of Virginia Tech. Report. Nov. 1993.

[Bere87a] A. D. Berenbaum, D. R. Ditzel and H. R. McLellan. Architectural Innovations in the CRISP Microprocessor. Proc. SPRING COMPCON87, pp. 91-95, Feb, 1987.

[Bene87b] A. D. Berenbaum, D. R. Ditzel and H. R. McLellan. Introduction to the CRISP Instruction Set Architecture. Proc. SPRING COMPCON87, pp. 86-90, Feb, 1987.

[Bern] David Bernstein, Martin E. Hopkins, and Michael Rodeh, International Business Machines Corporation. Speculative Load Instruction Rescheduler for a Compiler Which Moves Load Instructions Across Basic Block Boundaries While Avoiding Program Exceptions. United States Patent 5526499. http://

www.patents.ibm.com.

[Blic92] David S. Blickstein, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Jr., Kent D. Glossop, Richard B. Grove, Steven O. Hobbs and William B. Noyce. The GEM Optimizing Compiler System. Digital Technical Journal Digital Equipment Corporation, Vol. 4 No. 4, pp. 121-136, 1992.

[Brad91] David G. Bradlee, Susan J. Eggers and Robert R. Henry. The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy. Proc. 18th Intl. Symp. Computer Architecture, pp. 330-339, May. 1991.

[Brig92] Preston Briggs. Register Allocation via Graph Coloring.. Rice University, Houston, Texas, USA Tech. Report, 1992.

[Brun91] Richard A. Brunner. The VAX Architecture Reference Manual. Digital Press, Burlington, MA, 1991.

[Burg97] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin, Madison Tech. Report. June, 1997.

[Carm00] Doug Carmean. Inside the Pentium 4 Processor Micro-architecture. Intel Developer Forum Slides, August 24, 2000.

[Case92] Brian Case and Michael Slater. DEC Enters Microprocessor Business with Alpha. Microprocessor Report, Vol. 6 No. 3, pp. 24. March, 1992.

[Chai81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins and Peter W. Markstein. Register Allocation Via Coloring. Computer Languages, Vol. 6 No. 1, pp. 47-57, 1981.

[Chai82] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. Proc. SIGPLAN ‘82 Symp. Compiler Construction, pp. 98-105, 1982.

[Chen91] W. Y. Chen, P. P. Chang, T. M. Conte and W. W. Hwu. The Effect of Code Expanding Optimizations on Instruction Cache Design. Center for Reliable and High-Perform ance Computing, University of Illinois, Urbana-Champaign Tech. Report CRHC-91-17. May 1991.

[Chiu91] T.-C. Chiueh. An Integrated Memory Management Scheme for Dynamic Alias Resolution. Proc., Supercomputing ’91: Albuquerque, New Mexico, November 18-22, 1991, pp. 682-691, Aug. 1991.

[Chow84] Frederick Chow and John Hennessy. Register Allocation by Priority-based Coloring. ACM SIGPLAN Notices, pp. 222-232, June, 1984.

[Chow88] Fred C. Chow. Minimizing Register Usage Penalty at Procedure Calls. SIGPLAN ’88 Conf. Programming Language Design and Implementation, pp. 85-94, July 1988.

[Chow90] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. ACM Transactions Programming Languages and Systems, Vol. 12 No. 4, pp. 501-536, 1990.

[Coop88] K. D. Cooper and K. Kennedy. Interprocedural Side-Effect Analysis in Linear Time. ACM SIGPLAN Notices, Vol. 23 No. 7, pp. 57-66. July 1988.

[Coop97] Keith Cooper and John Lu. Register Promotion in C Programs. Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97), pp. 308-319, June, 1997.

[Coop98a] Keith D. Cooper and Timothy J. Harvey. Compiler-Controlled Memory. Eighth Intl. Conf. Architectural Support for Programming Languages and Operating Systems, pp. 100-104, Oct, 1998.

[Cruz00] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero and Nigel P. Topham. Multiple-Banked Register File Architectures. Proc. 27th Intl. Symp. Computer Architecture, pp. 316-325, June 2000.

[Dahl94] Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. Proc. 27th Intl. Symp. Microarchitecture, pp. 100-104, Nov, 1994.

[DEC83] Digital Equipment Corporation. The PDP-11 Architecture Handbook. Digital Press, Burlington, MA, 1983.

[Deif98] Keith Deifendorff. K7 Challenges Intel. Microprocessor Report, Vol. 12 No. 14 October 26, 1998.

[Ditz87a] D. R. Ditzel, H. R. McLellan and A. D. Berenbaum. Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor. Proc. Second Intl. Conf. Architectural Support for Programming Languages and Operating Systems-ASPLOSII, pp. 158-163, Oct, 1987.

[Ditz87b] David R. Ditzel, Hubert R. McLellan and Alan D. Berenbaum. The Hardware Architecture of the CRISP Microprocessor. Proc. 14th Intl. Symp. Computer Architecture, pp. 309-319, June, 1987.

[Diet88] H. Dietz and C.-H. Chi. CRegs: A New Kind of Memory for Referencing Arrays and Pointers. Proc., Supercomputing '88: November 14--18, 1988, Orlando, Florida, pp. 360-367, Jan, 1988.

[Ditz82] David R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. Proc. Symp. Architectural Support for Programming Languages and Operating Systems, pp. 48-56, March, 1982.

[EDG] http://www.edg.com.

[Emam94] Maryam Emami, Rakesh Ghiya and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. Proc. SIGPLAN '94 Conf. Programming Lanugage Design and Implementation, pp. 242-256, 1994.

[Emer84] J. S. Emer and D. W. Clark. A Characterization of Processor Performance in the VAX-11/780. Proc. 11th Symp. Computer Architecture, pp. 301-310, June, June 1984.

[Fark96] Keith I. Farkas, Norman P. Jouppi and Paul Chow. Register File Design Considerations in Dynamically Scheduled Processors. Proc. Second Intl. Symp. High Performance Computer Architecture, pp. 186-192, Jan. 1996.

[Flau97] Kris Flautner and David Greene. Optimization of MIRV Programs: Application of structural dataflow. December 11, 1997. Available at [MIRV].

[Flau98a] Krisztian Flautner, Gary S. Tyson and Trevor N. Mudge. MirvSim: A high level simulator integrated with the Mirv compiler. Proc. 3rd Workshop Interaction between Compilers and Computer Architectures (INTERACT-3), Oct. 1998.

[Flau98b] Kris Flautner, David Greene, and Trevor Mudge. MirvKit: A framework for compiler and computer architecture research. May 11, 1998. Available at [MIRV].

[Flau98c] Krisztián Flautner, David Greene, Matthew Postiff, David Helder, Charles Lefurgy, Peter Bird, and Trevor Mudge. Design of a High Level Intermediate Representation for Attribute-based Analysis. October 9, 1998. Available at [MIRV].

[Gall94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. ACM SIGPLAN Notices, Vol. 29 No. 11, pp. 183-193. Nov. 1994.

[Gamm95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Longman, 1995.

[GCC] http://gcc.gnu.org.

[GHS] http://www.ghs.com.

[Good96] D. W. Goodwin and K. D. Wilken. Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. Software Practice and Experience, Vol. 26 No. 8, pg. 929. Aug, 1996.

[Gonz97] Antonio Gonzalez, Mateo Valero, Jose Gonzalez and T. Monreal. Virtual Registers. Proc. Intl. Conf. High-Performance Computing, pp. 364-369, 1997.

[Gonz98] Antonio Gonzalez, Jose Gonzalez and Mateo Valero. Virtual-Physical Registers. Proc. 4th Intl. Symp. High-Performance Computer Architecture (HPCA-4), pp. 175-184, Feb. 1998.

[GPROF] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982.

[Great] Great Microprocessors of the Past and Present. http://www.microprocessor.sscc.ru/great.

[Green98] David Greene. Profile-Guided Function Specialization in the MIRV Compiler. September 15, 1998. Available at [MIRV].

[Gwen94a] Linley Gwennap. Digital Leads the Pack with 21164. Microprocessor Report, Vol. 8, No. 12. September 12, 1994.

[Gwen94b] Linley Gwennap. PA-8000 Combines Complexity and Speed. Microprocessor Report, Vol. 8, No. 15. November 14, 1994.

[Gwen95] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. Microprocessor Report, Vol. 9 No. 2 February 16, 1995.

[Gwen96] Linley Gwenapp. Digital 21264 Sets New Standard. Microprocessor Report, Vol. 10, No. 14. October 28, 1996, pp. 11-16.

[Gwen98] Linley Gwennap. Mendocino Improves Celeron. Microprocessor Report, Vol. 12, No. 11, August 24, 1998.

[Half00a] Tom R. Halfhill. SiByte Reveals 64-Bit Core for NPUs. Microprocessor Report, Vol. 14 No. 6 June 2000.

[Half00b] Tom R. Halfhill. Transmeta Breaks x86 Low-Power Barrier. Microprocessor Report, Vol. 14 No. 2 February 2000.

[Hall96] Mary W. Hall, Jennifer M. Anderson, Saman p. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. Computer, Vol. 29 No. 12, pg. 84. Dec, 1996.

[Harb82] Samuel P. Harbison. An Architectural Alternative to Optimizing Compilers. Proc. Symp. Architectural Support for Programming Languages and Operating Systems, pp. 57-65, March, 1982.

[Henn96] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Mateo, CA, 1996.

[Heggy90] B. Heggy and M. L. Soffa. Architectural Support for Register Allocation in the Presence of Aliasing. Proc., Supercomputing '90: November 12--16, 1990, New York Hilton at Rockefeller Center, New York, New York, pp. 730-739, Feb, 1990.

[Hook97] R. J. Hookway and M. A. Herdeg. DIGITAL FX!32: Combining Emulation and Binary Translation. Digital Technical Journal Digital Equipment Corporation, Vol. 9 No. 1, pg. 3, 1997.

[IA6499] Intel IA-64 Application Developer's Architecture Guide. May 1999. Order Number: 245188-001. Available at http://developer.intel.com/design/ia64/devinfo.htm.

[IBM98] IBM. Enterprise Systems Architecture/390 Principles of Operation. IBM, Poughkeepsie, New York, 1998.

[Impact] http://www.crhc.uiuc.edu/Impact.

[Inte96] Intel Corporation. Penium Pro Family Developer's Manual. Volume 2: Programmer's Reference Manual. Order Number 242691. 1996.

[Kell75] Robert M. Keller. Look-Ahead Processors. ACM Computing Surveys, Vol. 7 No. 4, pp. 177-195. Dec, 1975.

[Kenn] H. Roland Kenner, Alan Karp, and William Chen, Institute for the Develoment of Emerging Architecture, L.L.C. Method and apparatus for implementing check instructions that allow for the reuse of memory conflict information if no memory conflict occurs. United States Patent 5903749. http://www.patents.ibm.com.

[Kilb62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan and F. H. Sumner. One-level Storage System. IRE Transactions Electronic Computers, Vol. EC-11 No. 2, pp. 223-235. April, 1962.

[Kiyo] Tokuzo Kiyohara, Wen-mei W. Hwu; William Chen, Matsushita Electric Industrial Co., Ltd., and The Board of Trustees of the University of Illinois. Memory conflict buffer for achieving memory disambiguation in compile-time code schedule. United States Patent 5694577. http://www.patents.ibm.com.

[Kiyo93] T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik and W.-M. Hwu. Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. Proc. 20th Intl. Symp. Computer Architecture, pp. 247-256, May, 1993.

[Klai00] Alexander Klaiber. The Technology Behind Crusoe^{TM} Processors. Transmeta Corporation. January 2000.

[Kong98] Timothy Kong and Kent. D. Wilken. Precise Register Allocation for Irregular Architectures. Proc. ACM SIGPLAN’98 Conf. Programming Language Design and Implementation (PLDI), pp. 297-307, 1998.

[Kuck78] David J. Kuck. The Structure of Computers and Computations. John Wiley & Sons, Pittsburgh, Pennsylvania, 1978.

[LCC] http://www.cs.princeton.edu/lcc.

[Lo98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu and Peng Tu. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. Proc. ACM SIGPLAN’98 Conf. Programming Language Design and Implementation (PLDI), pp. 26-37, 1998.

[Lo99] Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Heny M. Levy, Dean M. Tullsen. Software-Directed Register Deallocation for Simultaneous Multithreaded Processors. IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 9, September 1999, pp. 922-933.

[Lone61] William Lonergan and Paul King. Design of the B5000 system. Datamation, Vol. 7 No. 5, pp. 28-32. May, 1961.

[Loza95] Luis A. Lozano C. and Guang R. Gao. Exploiting Short-Lived Variables in Superscalar Processors. Proc. 28th Intl. Symp. Microarchitecture, pp. 292-302, Nov. 1995.

[Lu98] John Lu. Interprocedural Pointer Analysis for C. Rice University, Houston, Texas, USA Tech. Report. April, 1998.

[MIRV] http://www.eecs.umich.edu/mirv.

[Mahl92a] Scott A. Mahlke, William Y. Chen, Pohua P. Chang and Wen-mei W. Hwu. Scalar Program Performance on Multiple-Instruction-Issue Processors with a Limited Number of Registers. Proc. 25th Hawaii Intl. Conf. System Sciences, pp. 34-44, Jan 6-9, 1992.

[Mahl92b] 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. Proc. Supercomputing '92: Minneapolis, Minnesota, November 16-20, 1992, pp. 808-817, Aug. 1992.

[Mart97] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting Dead Value Information. Proc. 30th Intl. Symp. Microarchitecture (MICRO'97), Dec. 1997.

[Monr99] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez and V. Vinals. Delaying Physical Register Allocation through Virtual-Physical Registers. Proc. 32nd Intl. Symp. Microarchitecture, pp. 186-192, Nov. 1999.

[Moor65] Gordon E. Moore. Cramming more components onto integrated circuits. Electronics, Vol. 38 No. 8, pp. 114-117. April, 1965.

[Moud93] M. Moudgill, K. Pingali and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. Proc. 26th Intl. Symp. Microarchitecture (MICRO'93), pp. 202-213, Dec. 1993.

[MDR99] Chart Watch: Workstation Processors. Microprocessor Report, Vol. 13 No. 1, pp. 31. Jan, 1999.

[MDR99b]Chart Watch: Workstation Processors. Microprocessor Report, Vol. 15, No. 14. October 25, 1999.

[Much97] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[Nico89] Alexandru Nicolau. Run-Time Disambiguation: Comping with Statically Unpredictable Dependencies. IEEE Transactions Computers, Vol. 38 No. 5, pp. 663-678. May, 1989.

[Nowa92] S. Nowakowski and M. T. O'Keefe. A CRegs Implementation Study Based on the MIPS-X RISC Processor. Intl. Conf. Computer Design, VLSI in Computers and Processors, pp. 558-563, Oct, 1992.

[PTSC] http://www.ptsc.com/psc1000/overview.html.

[Patt81] David A. Patterson and Carlo H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. Proc. 8th Intl. Symp. Computer Architecture, Vol. 32 No. CS-93-63, pp. 443-457. Nov. 1981.

[Post99] Matthew A. Postiff and Trevor Mudge. Smart Register Files for High-Performance Microprocessors. University of Michigan CSE Technical Report CSE-TR-403-99, June 1999.

[Post00a] Matthew Postiff, David Greene, Charles Lefurgy, Dave Helder, Trevor Mudge. The MIRV SimpleScalar/PISA Compiler. University of Michigan CSE Technical Report CSE-TR-421-00, April 2000. Available at [MIRV].

[Post00b] Matthew Postiff, David Greene, and Trevor Mudge. Exploiting Large Register Files in General Purpose Code. University of Michigan Technical Report CSE-TR-434-00, October 2000. Available at [MIRV].

[Post00c] Matthew Postiff, David Greene, Greene and Trevor Mudge. The Store-Load Address Table and Speculative Register Promotion. Proc. 33rd Annual Intl. Symp. Microarchitecture (Micro33), Monterrey, CA. December 10-13, 2000, pp. 235-244.

[Rein98] August G. Reinig. Alias Analysis in the DEC C and DIGITAL C++ Compilers. Digital Technical Journal, Vol. 10 No. 1, pp. 48-57. Dec, 1998.

[Rixn00] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujival J. Kapasi and John D. Owens. Register Organization for Media Processing. Proc. 6th Intl. Symp. High-Performance Computer Architecture, pp. 375-386, Jan. 2000.

[Sast98] A. V. S. Sastry and Roy D. C. Ju. A New Algorithm for Scalar Register Promotion Based on SSA Form. Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI), pp. 15-25, 1998.

[Siew82] Daniel P. Siewiorek, C. G. Bell and A. Newell. Computer Structures: Principles and Examples. McGraw-Hill, Pittsburgh, Pennsylvania, 1982.

[Smit82] Alan Jay Smith. Cache Memories. ACM Computing Surveys, Vol. 14 No. 3, pp. 473-530. Sep, 1982.

[SGI] http://oss.sgi.com/projects/Pro64.

[Siem97] Siemens. TriCore Architecture Overview Handbook. Version 1.1. September 17, 1997.

[Sima00] Dezso Sima. The Design Space of Register Renaming Techniques. IEEE Micro, Vol. 20 No. 5, pp. 70-83. Sep/Oct 2000.

[Site79] Richard L. Sites. How to Use 1000 Registers. Proc. 1st Caltech Conf. VLSI, pp. 527-532, Jan, 1979.

[Site92] Richard L. Sites. The Alpha Architecture Reference Manual. Digital Press, Burlington, MA, 1992.

[Srin98] Srikanth T. Srinivasan and Alvin R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. Proc. 31st Intl. Symp. Microarchitecture, pp. 148-159, Nov. 30-Dec 2, 1998.

[SUIF] http://suif.stanford.edu.

[Swen88] John A. Swenson and Yale N. Patt. Hierarchical Registers for Scientific Computers. Proc. Intl. Conf. Supercomputing, pp. 346-353, July 1988.

[SysV91] UNIX System Laboratories Inc. System V Application Binary Interface: MIPS Processor Supplement. Unix Press/Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Taka63] S. Takahashi, H. Nishino, K. Yoshihiro and K. Fuchi. System Design of the ETL Mk-6 Computers. Information Processing 1962, Proc. IFIP Congress, pg. 690, 1963.

[TI97] Texas Instruments. TMS320 DSP Development Support Reference Guide and Addendum. August 1997. Literature Number SPRU226.

[TM1K] Philips Electronics. TM1000 Preliminary Data Book. TriMedia Product Group, 8111 E. Arques Avenue, Sunnyvalue, CA 94088. 1997.

[Tami83] Yuval Tamir and Carlo H. Sequin. Strategies for managing the register file in RISC. ACM Transactions Computer Systems, Vol. 32 No. 11, pp. 977-988. Aug. 1983.

[Toma67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. IBM Journal Research and Development, Vol. 11 No. 1, pp. 25-33. Jan, 1967.

[Trimaran] http://www.trimaran.org.

[Tyso97] Gary S. Tyson and Todd M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. Proc. 30th Intl. Symp. Microarchitecture, pp. 218-227, Dec, 1997.

[Vand00] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. ACM Computing Surveys, Vol. 32, No. 2, June 2000, pp. 174-199.

[Vlao00] Stevan Vlaovic, Edward S. Davidson and Gary S. Tyson. Improving BTB Performance in the Presence of DLLs. Proc. 33rd Intl. Symp. Microarchitecture (MICRO 2000), pp. 77-86, Dec. 10-13, 2000.

[Wall86] David W. Wall. Global Register Allocation at Link Time. Proc. SIGPLAN’86 Symp. Compiler Construction, pp. 264-275, July, 1986.

[Wall88] David W. Wall. Register Windows vs. Register Allocation. ACM SIGPLAN Notices, Vol. 23 No. 7, pp. 67-78. July 1988.

[Weav94] David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. Sparc International and PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[Wilk65] M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. IEEE Transactions Electronic Computers, Vol. EC-14 No. 2, pp. 270-271. April, 1965.

[Wils95] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. Proc. ACM SIGPLAN'95 Conf. Programming Language Design and Implementation (PLDI), pg. 1, 1995.

[Wing] Malcom J. Wing and Edmund J. Kelly, Transmeta Corporation. Method and apparatus for aliasing memory data in an advanced microprocessor. United States Patent 5926832. http://www.patents.ibm.com.

[Yeag96] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. IEEE Micro, Vol. 16 No. 2, pp. 28-40. April, 1996.

[Yung95a] Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Intl. Conf. Computer Design, pp. 307-312, Oct, 1995.

[Yung95b] Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Sun Microsystems Laboratories Tech. Report. June, 1995.

[Zala00] Javier Zalamea, Josep Llosa, Eduard AyguadE and Mateo Valero. Two-level Hierarchical Register File Organization for VLIW Processors. European Center of Parallelism of Barcelona Technical Report UPC-CEPBA-2000-20. June 27, 2000.