Pseudo-Vector Machine For Embedded Applications

by

Lea Hwang Lee

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Science and Engineering)

at the
The University of Michigan
2000

Doctoral Committee:

Prof. Trevor Mudge, Chairperson
Prof. Richard Brown
Prof. Ed Davidson
Prof. Marios Papaefthymiou
Prof. Karem Sakallah

“We are moving into a third stage of computing. In the first, the mainframe world, there was one computer for many people. In the second, the PC world, there was a computer for each person. In the next stage there will be many computing devices for each person...”

Roy Want,
Palo Alto Research Center, Xerox Corp.,
Palo Alto, CA.
Source: PC Week Online, January 3, 2000

To My Mother,
Chern, Der-Shin,
And
My Sister,
Lee, Deek Ann,

For their unfailing care and support.

ACKNOWLEDGEMENTS

I joint the M-CORE™ Technology Center (MTC), Motorola Incorp., Austin, Texas, as a summer intern in 1994. At that time, the group (under a different name) had just embarked on developing a new ISA for targeting mid-to-low end embedded markets. I spent the next two years (1995 and 1996) working and traveling between Austin, Texas and Ann Arbor, Michigan. I became a full-time employee towards the end of 1996.

This dissertation work is not formally nor directly funded by any organization. However, I did receive a lot of assistance from the MTC. In particular, they had given me access to various software tools and benchmark programs. For a brief period of time, they had also kept me on the payroll while I was working full-time on my dissertation - what a perfect way to fund a research project.

I am deeply indebted to my manager, John Arends, for all the constant guidance he had provided to me. From day one, he has always been there to renter his help, technically or otherwise.

I am very grateful to my co-worker and my best friend, Jeff Scott. He is one of the most talented, energetic, motivated and creative engineer I have ever met. I am also very grateful to Bill Moyer for all his technical assistance and insightful advises. His insights has always made my “research life” much more interesting and challenging.

I am also deeply indebted to Prof. Trevor Mudge for all his technical assistance and advises, without which, this dissertation work would be impossible.

Lastly, but not the least, I would like to thank the entire MTC team, for all their support they have given to me. I feel honored, to have such a great opportunity to work with this talented and motivated group of people.

PREFACE

The focus of this dissertation is on designing low-cost, low-power and high performance processors for mid-to-low end mobile embedded applications. In particular, the main goal of this work is to *explore ways to add a minimum amount of hardware to a single issued machine to improve its performance on critical loop executions* - since many of these applications spend a significant amount of their execution time on a handful of critical loops. Improve the performance on these loops provides the biggest bang for the buck.

This dissertation borrows many existing architectural ideas from vector processors and DSP processors, and combines them into a single execution model.

Vector processing paradigm is well known for its excellent cost/performance trade-off. The processing paradigm proposed in this dissertation, called the *pseudo-vector machine*, exploits, as much as possible, the low-cost, low-power and high performance aspects of vector processing paradigm.

As we will see later in this dissertation, the characteristics of the critical loops found in our benchmarks vary greatly, from *highly vectorizable*, to *difficult* (and *costly*) *to vectorize*, to *impossible to vectorize*.

For example, a loop that performs a vector operation described by C[i] = A[i] * B[i], for i=0,...,n-1, for some vectors A, B and C of length n, is a highly vectorizable loop. A loop that performs a vector operation described by C[i] = (A[i]>B[i])? $A[i]^2$: A[i]+B[i], for i=0,...,n-1, is more difficult (or costlier) to vectorize.

The vector arithmetic represented by some hard-to-vectorize loops, in this work, is called *pseudo-vector arithmetic* (PVA). For this type of loops, the DSP's style of processing, which focuses on optimizing the program loop executions, is more suitable. These optimization techniques include: (i) using the "while" or "repeat" instructions to remove loop control overheads; (ii) using data streaming buffers to remove overhead associated with constant-stride

memory operations.

The pseudo-vector machine proposed in this dissertation can execute two types of vector arithmetic: a “true” vector arithmetic and a “pseudo” vector arithmetic, as described in the preceding paragraph. Depending on the type of loops, the machine sometime behaves like a vector processor; it sometime behaves like a DSP processor. The compilers, for this machine, decide which execution modes to use for each critical loop.

In addition, this machine uses a single datapath to execute all the vector arithmetic as well as the scalar portion (the non-loop portion) of the program code - an efficient reuse of the hardware resources.

TABLE OF CONTENTS

CHAPTER I

CHAPTER II

CHAPTER III

CHAPTER IV

CHAPTER VI

LIST OF FIGURES

LIST OF TABLES

FREQUENTLY USED M-CORE INSTRUCTIONS

The following contains brief descriptions of M-CORE instructions that are frequently used throughout this dissertation.

In this ISA, an instruction typically has the following format: ops rx, ry, where rx and ry are source registers; and rx is also the destination register [MCORE98].

Table 1: Frequently Used M-CORE Instructions

Mnemonic	Description	Example
ldb, ldh, ldw stb, sth, stw	Load byte; load halfword; load word Store byte; store halfword; store word	ldw r6, (r2) stb r7, 4(r3)
bf, bt	Branch if c-bit clear; branch if c-bit set	bt TARGET
cmplt cmpne cmphs cmpnei	Compare less than, set c-bit if true Compare not equal, set c-bit if true Compare higher or same, set c-bit if true Compare not equal immediate, set c-bit if true	cmplt r4, r7 cmpnei r6, 3

add, addi,
sub, subi,
rsub
	Add; add immediate
	Subtract (rx = rx - ry); subtract immediate
	Reverse subtract (rx = ry - rx)
		add r6, r8
		subi r8, 2
		rsub r9, r2

decne declt	Decrement, then set c-bit if not equal 0 Decrement, then set c-bit if less than 0	decne r1 declt r1
mov, movt	Move; move if c-bit set	mov r3, r6
lsr, lsri lsl, lsli asri	Logical shift right; logical shift right immediate Logical shift left; logical shift left immediate Arithmetic shift right immediate	lsr r7, r5 lsli r7, 8 asri r7, 8
or, xor	Logical or; exclusive-or	or r2, r3
clrt	Clear register if c-bit set	clrt r8
mul	multiply	mul r3, r4
ixh	Index halfword (rx = rx + (ry <<1))	ixh r4, r1
tst	Test with zero (clear c-bit if (rx & ry) == 0; set otherwise)	tst r4, r5
zextb	Zero extent least significant byte	zextb r3
lrw	Load relative word: load a word from DATA_LABEL into rx	lrw r2, [DATA_LABEL]
mtcr, mfcr	Move to control register from general purpose register; move from control register to general purpose register	mtcr r3, SSR

Trademarks

M-COREis a trademark of Motorola Corporation.

i500plus is a trademark of Motorola Corporation.

Palm VII is a trademark of 3Com Corporation.

Palm.Net is a trademark of 3Com Corporation.

SmartPhone is a trademark of Neopoint Incorporation.

WebBank is a trademark of Leonia Bank Corporation.

SHARC is a trademark of Analog Devices Corporation.

TMS320Cxx is a trademark of Texas Instrument Incorporation.

PowerPC AltiVec is a trademark of Motorola Corporation.

Cray-1 is a trademark of Cray Research Incorporation.

IA-64 is a trademark of Intel Corporation.

TriCore is a trademark of Siemens Incorporation.

Crusoe is a trademark of Transmeta Corporation.

Code Morphing is a trademark of Transmeta Corporation.

StarCore is a trademark of Motorola Corporation and Lucent Technologies.

CHAPTER 1

INTRODUCTION: LOW-COST, LOW-POWER DESIGNS FOR EMBEDDED APPLICATIONS

Low-cost, low-power designs have been gaining importance in microprocessor systems primarily due to increasingly wide spread use of portable and handheld applications. These applications are also known as *mobile applications*.

Mobile computing and mobile applications, for the purpose of this work, refer to computing systems for consumers' portable and handheld applications that include pagers, cellular phones, personal digital assistants (PDA), global positioning systems (GPS), etc. These applications are powered by a battery system that has a limited energy storage capacity.

In this Chapter, we will examine the importance of mobile computing systems and their future trends. We will also examine some of the important characteristics and requirements of such systems. We will give an overview of our proposed execution model, called the *pseudo-vector machine*, for mobile applications. We will then examine the strength of vector processing and their suitability for such systems.

1.1 The World Of Mobile Computing

The Personal Computer (PC) market has been enjoying tremendous growth in the past two decades. The growth in this market, in terms of profit margins, has recently been slowed down. The sharpest growth has been gradually shifting from the low volume and high profit margin, to the high volume and low profit margin PC market - a sign of a maturing market. The market as a whole will continue to grow, primarily being sustained by: (i) the general break down of the world's traditional trading boundaries, opening up the world's mass markets; and by (ii) the rapid advances and growth in the internet.

We are entering an era of *personal communication*. Early nineties marks the beginning of this era where we saw the rapid growth in computing, pagers, cellular phones and web-based mobile applications. This era is characterized by the needs of an individual to acquire the information and knowledge he or she needs, at any time, at anywhere, precisely and instantly.

The personal communication era was enabled by the adaptation and popularization of the internet. With the society becomes increasingly mobile, accessing to the internet via the traditional workstations or desktop computers can no longer satisfy the changing needs of the consumers. These needs range from stock trading, accessing to the weather forecasts, to locating a restaurant. Mobile computing serves these needs. The growth of the internet will continue to benefit the mobile computing market.

3Com Corp. has been offering Palm VII™ Connected Organizer since 1998. In conjunction with the Palm.Net™ internet access service provided by 3Com, a user can use Palm VII to send/receive emails, obtain stock quote, sports scores, check flight information or the weather, etc. [PALMVII98].

Motorola Incorp., through Nextel Communications Incorp., has been offering iDEN i500plus™ multi-service digital wireless phone. This WML (Wireless Markup Language)-compliant phone features Internet microbrowser, two-way e-mail and wireless modem functionality. A user of this phone can check news, weather and stocks, find phone number and address information, and even get driving directions [IDEN99].

Nokia Corp. and Palm Computing Inc. (a 3Com company) have recently announced a broad joint development and licensing agreement to create a new pen-based product line that integrates telephony with data applications, personal and professional information management [PALMVII99].

Neopoint Inc. has recently announced its availability of its Smartphone™ (through Sprint) - a cellular phone with limited web browsing capabilities [NEOPOINT99a, NEOPOINT99b].

Leonia Bank of Finland will be offering its customers online banking services, WebBank™, through the WAP (Wireless Application Protocols) phones, in the spring of 2000. These services are based on Public Key Infrastructure (PKI), digital signatures and strong encryption. The bank will identify the customer with digital signatures located on a SIM card of a mobile phone. Customers using digital signatures do not need cumbersome passwords anymore. With digital signatures mobile phone banking will be a lot easier and more secure than before [LEONIA00].

Very soon, we will be seeing all the consumers' mobile applications, including email, fax, pager, personal organizer, cellular phone, GPS, stock quote, weather forecast and other web-based applications integrated into a single, lightweight, device that everyone carries.

1.2 Important Characteristics Of Mobile Systems

One of the important factors that decides whether a portable product can succeed in the market place is "how long can the product operate before its battery needs to be replaced or recharged". Other important factors are the system costs and performance.

Perhaps to the surprise of some readers, in many mobile systems, improving the performance can often help solve the "energy problems." Unfortunately, the "low-power objective" for mobile computing systems is not a straight forward concept. To begin with, the low power design goal for such a system is not to maximize the battery life, but to maximize the number of operations performed per battery discharge.

In order to further understand the "power problem" in mobile computing, we will first need to understand some characteristics of a mobile computing system.

1.2.1 Alternate Operating Modes

To conserve energy, power-down techniques are widely employed in mobile computing systems, at all levels of design hierarchies. A mobile application typically operates alternatively between two operating modes: (i) burst mode (or active mode), where active computations are performed; and (ii) power-down mode (or sleep mode), where the system is asleep waiting for a new computational event to occur. A paging system, for example, is only awaken to process an incoming message, and is put back to sleep once the computation is completed.

Figure 1.1 shows an example of the instantaneous power of a mobile computing system, going through a series of burst and sleep modes.

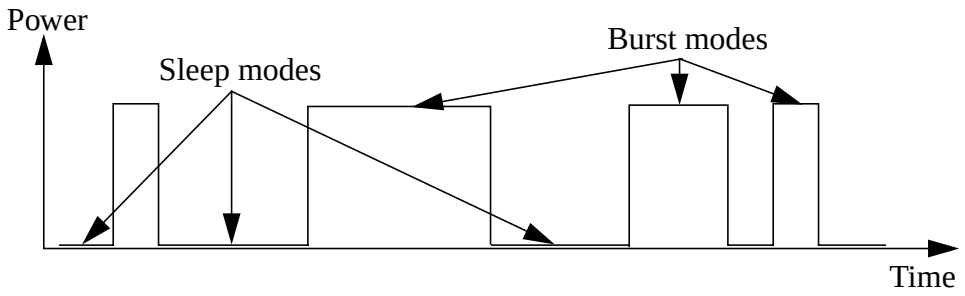


Figure 1.1: Instantaneous Power Of A Mobile Computing System

A mobile computing system typically consists of a digital subsystem, among with other subsystems. A cellular system, for example, these other subsystems include receiver/transmitter subsystem, the display subsystem, the keypad subsystem, etc.

From the energy consumption point of view, if the digital subsystem consumes only a small fraction of the overall system power, then it will be advantageous to design that subsystem to improve the overall system performance. By shortening the time the system is in burst mode (thus shortening the time the system is consuming high power), the overall energy consumption can be reduced.

Figure 1.2 shows the instantaneous power of two competing designs operating through a "sleep-burst-sleep" sequence. Design A has higher performance with slightly higher power during burst mode, while Design B has lower performance with slightly lower power during burst mode. Design A completes the computational task earlier and immediately puts the system to sleep. Since energy consumption is the integration of power over time (given by the area under the power-time curve), the overall energy consumption of the system using Design A, in this case, is lower than that of Design B.

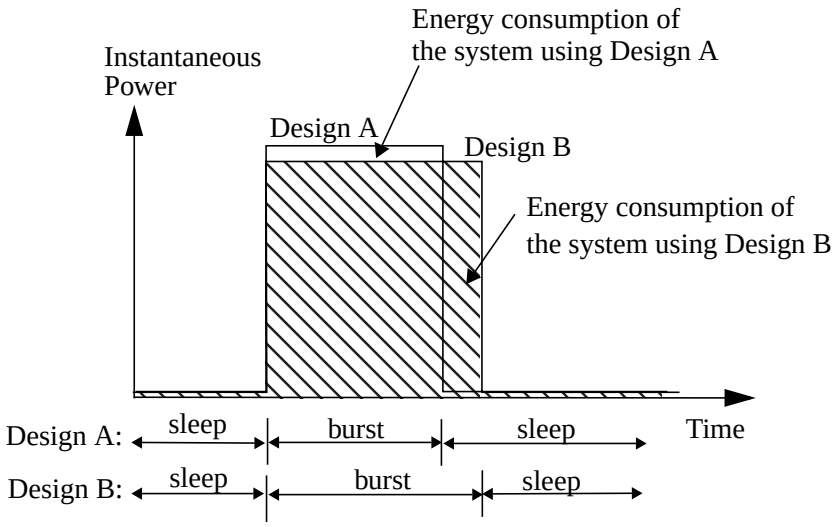


Figure 1.2: Energy versus power consumption

1.2.2 Performance vs. Instantaneous Power

From the above example, it is arguable that *if a digital subsystem (microprocessor included) consumes only a small fraction of the overall system power, then it is advantageous to design the subsystem to improve the overall system performance.* That is, we solve the "power problem" by improving the performance, possibly at the expense of slightly higher power consumption during burst mode.

The battery systems are, unfortunately, not ideal. They possess internal resistance. If higher power is continuously drawn from these batteries, then less useful energy will be available to the rest of the system due to this internal resistance. To reduce such dissipations, one would argue that we should operate a mobile system at a lower power level, possibly at the expense of performance. That is, we solve the “power problem” by lowering the power consumption during burst modes [Martin99, Surampudi99].

By now it should become clear that as far as the digital subsystem is concerned, to achieve the “low-energy” object, one needs to perform complex trade-offs that involve the operating voltage and frequency of the digital subsystem, as well as the system environments which include the power consumption characteristics of other subsystems, and the battery subsystem itself.

In this dissertation work, we will focus on designing microprocessor cores for *mid to low-end, ultra-lightweight, embedded mobile applications*. *In these specific design environments, the digital subsystems often consume a small fraction of the overall power. We will thus adapt the “high performance” approach in solving the “power problem”, as described in Section 1.2.1.* In summary, the design goals in our design environments are (in order of importance):

1. Low cost (low chip area);
2. High performance;
3. Low power.

The above ordering has the following implications. Once an area budget is set for a specific application, we should then design the digital subsystem to improve the overall performance, as much as possible. In doing so, we can often (though not always) lower the overall energy consumption. Furthermore, any low-power design technique that can reduce power consumption during burst mode, should only be used *if* they do not degrade the overall performance.

Conversely, if a design technique that reduces power consumption of the digital subsystem during burst mode but degrades the system performance, then the overall energy consumption may reduce or increase, depending on the system power relative to the power consumption of the digital subsystem.

1.3 Execution Modes For Mobile Systems

At certain time, many mobile applications require the machines to perform some highly repetitive DSP functions. Large amount of instruction level parallelism (ILP) is present in these

applications. But at some other time, they require the machines to perform control intensive functions.

To address the needs for mobile computing, some desktop CPUs have been incorporating some DSP capabilities into their instruction set and designs [Massana99, Lexra99, etc.]. On the other hand, some DSP processors are now incorporating some general control functions into their designs [SHARC97,StarCore98].

Some systems use dual-core solutions to address this problem [DSP56654,etc.]. In these systems, one core performs all the control intensive functions; while the other core performs the specialized DSP functions. The two cores communicate through some communication channels, such as a shared memory. These systems often employ dual instruction streams, one for each execution cores. The high development costs associated with this approach could only be justified if the specialized market segment where the system is designed to, has sufficiently large market volume to amortize these non-recurring engineering costs.

1.4 Pseudo-Vector Machine - An Architectural Overview

In this dissertation, we will present a processing paradigm that is capable of executing in two modes: (i) a scalar execution mode for control functions; and (ii) a vector execution mode for exploiting the ILP that is present in these applications.

The vector execution mode can be further divided into a “true” vector mode and a “pseudo” vector mode. For loops that are highly vectorizable, the machine uses a “true” vector processing paradigm to process the loops. For loops that are difficult or impossible to vectorize, the machine uses a “pseudo” vector processing paradigm (similar to the DSP’s style of processing) to process the loop.

This machine executes all these modes on a single datapath using a single instruction stream. Each instruction in this stream can be classified as either a scalar instruction or a vector instruction. When a vector instruction is fetched and decoded, the machine enters a vector execution mode. The machine only exits the vector mode via a few pre-defined mechanisms. We will call this execution model the *pseudo-vector machine*.

The following are some important features of the pseudo-vector machine.

- The pseudo-vector machine has two major execution modes: scalar mode and vector mode.
- While in vector mode, this machine can perform two types of vector arithmetic: the *Canonical*

Vector Arithmetic (CVA) and the *Pseudo-Vector Arithmetic* (PVA). Correspondingly, there are two vector instructions: a CVA instruction and a PVA instruction. Figure 1.3 shows all the execution modes in this machine.

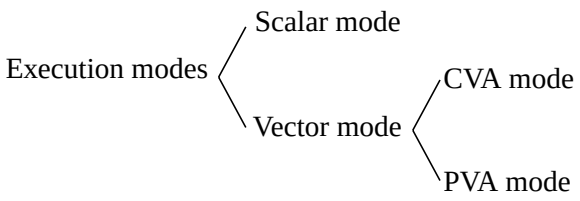


Figure 1.3: Various Execution Modes On The Pseudo-Vector Machine

- When executing in a vector mode, the vector instruction (CVA or PVA instruction) can optionally enable up to two input data streams from the memory (denoted as L_0 and L_1) and one output data stream to the memory (denoted as S).
- When the machine executes in a CVA mode, data are continuously streamed from the memory, processed by a chain of functional units and streamed back to the memory, in a highly pipelined fashion. The CVA mode represents a “true” vector processing paradigm.
- When the machine executes in a PVA mode, the corresponding assembly code consists of a PVA instruction followed by a *loop body*. The loop body is composed of multiple scalar instructions. The PVA instruction is very similar to the “DO UNTIL” or “REPEAT” instructions in the traditional DSP processors. A PVA instruction can optionally enable up to two constant-stride load operations (L_0 and L_1 streams) and one constant-stride store operation (S stream) to be automatically performed during loop executions. The PVA mode represents a “pseudo” vector processing paradigm similar to the DSP’s style of processing.
- The CPU architecture of the pseudo-vector machine is shown in Figure 1.4.
- When executing in a vector mode (CVA or PVA mode), the memory system can support up to two data reads, one data write and one instruction fetch in each cycle.
- In this machine, there are three independent on-chip memory modules, M0, M1 and TM. M0 supplies instructions and data, while M1 supplies data only. TM (temporary memory) is a small memory block used to store temporary vectors only.
- When executing program loops, a small *loop cache* is used to store the instructions.
- Within the execution core, there are two load units, L_0 and L_1, a store unit S, a register file Regfile and a general purpose functional unit P (see Figure 1.4).
- Besides performing the memory store operations, the S unit can also perform some simple, commutative arithmetic and logical functions, such as “add”, “and”, “or”, “xor”, etc.
- When executing in a scalar mode, the execution core behaves like a single-issued pipelined

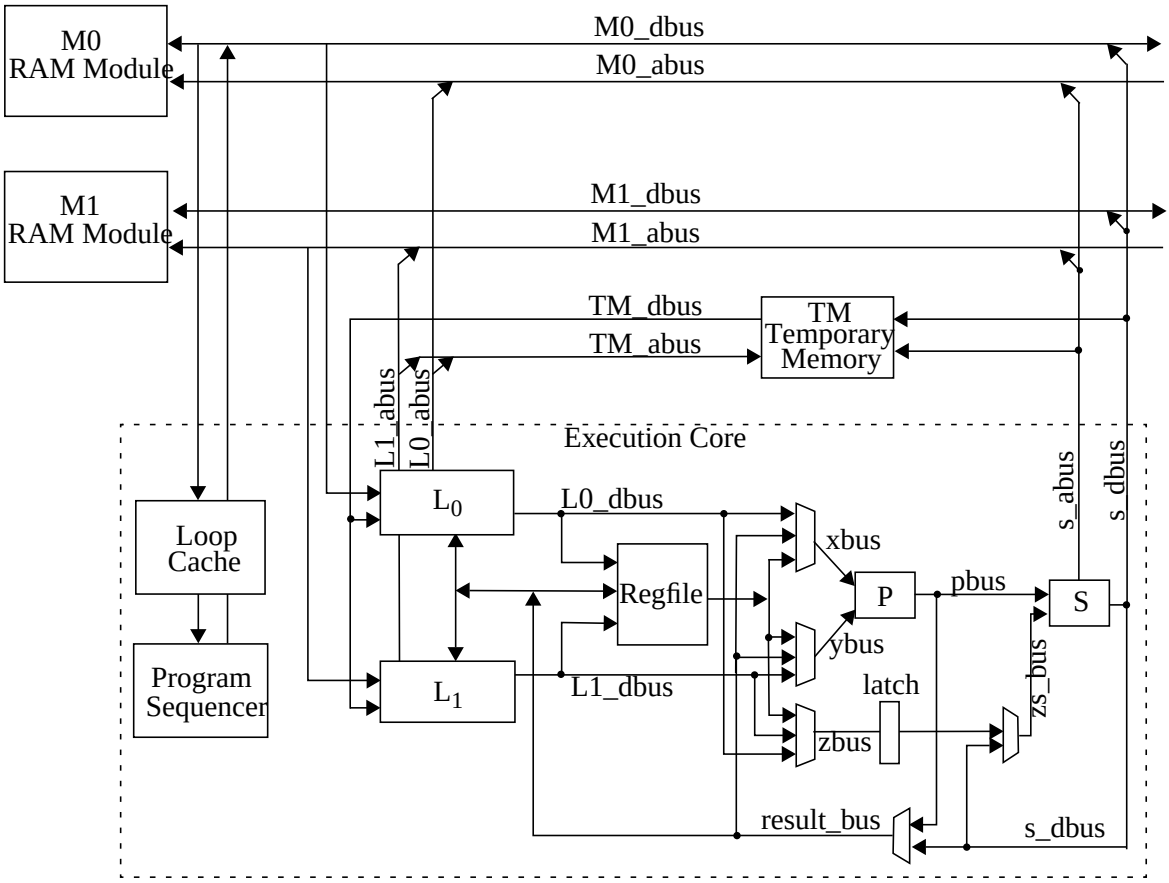


Figure 1.4: The CPU Architecture for the Pseudo-Vector Machine

machine. It uses the Regfile and the P unit for computation, and L_0 and S units for memory load and store operations, respectively.

- When executing in a PVA mode, the execution core also behaves like a single-issued pipeline machine, except that it can optionally enable up to: (i) two input data streams L_0 and L_1; and (ii) one output data stream S. In addition, the machine uses some special loop count register to remove execution overheads associated with branches and loop control mechanisms.
- When executing in a CVA mode, data are continuously streamed in from M0/M1/TM, through the L_0 and L_1 units, processed by the P and S units, and optionally streamed back to M0/M1/TM. When executing in this mode, two distinct scalar arithmetic, p_op and s_op, can be simultaneously performed at P and S, respectively.
- There are three types of CVA executions: *Compound CVA*, *Reduction CVA* and *Hybrid CVA*. The data dependency graphs for these three types of executions are shown in Figure 1.5. In this Figure, operands X, Y and Z can source from input streams L_0, L_1 or a scalar s. The outputs of these executions can be written to an output stream S or to a scalar destination s.
- In this datapath, all scalar and vector executions use the same P unit. This means that many

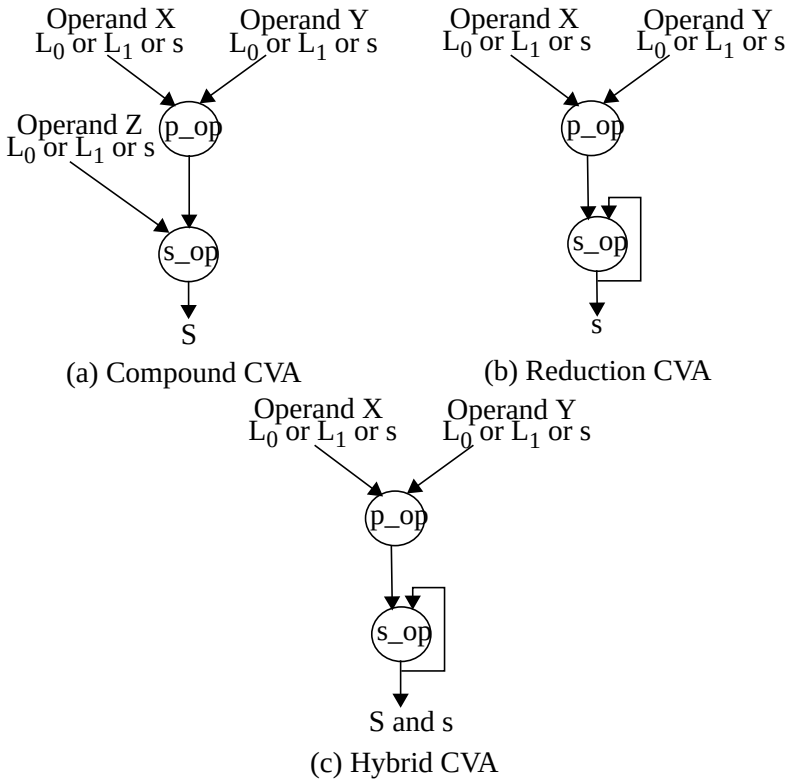


Figure 1.5: Dependency Graphs for Three Types of CVA

arithmetic functions that are available to the scalar executions are also available to the vector executions.

1.5 The Strength of Vector Processing

In this Section, we will attempt to motivate the strength of vector processing, in the context of low-cost, low-power embedded computing environments. Readers who are familiar with vector processing may skip this Section entirely. Consider the following example.

Example 1.1:

The following program loop performs an element-wise multiplication on two vectors. In vector form, it is performing: C[i]= A[i] * B[i], for all i. The data dependency graph of this program loop is shown in Figure 1.6.

```
L1:
addi    r2,2            // update stride value
ld.h    r7,(r2)         // load A[i]
addi    r3,1            // update stride value
ld.b    r6,(r3)         // load B[i]
mul     r7,r6           // multiply A[i] * B[i]
```

```
addi    r4,2            // update stride value
st.h    r7,(r4)         // store C[i]
decne   r1              // decrement loop index r1
                        // set c bit if r1 not equals zero
bt      L1              // backward branch if c bit set
```

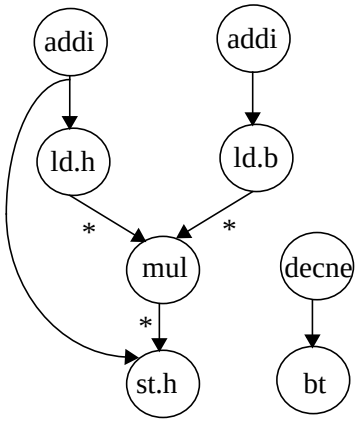


Figure 1.6: Data dependency graph for Example 1.1

In this example, intermediate values are produced and then consumed immediately. They are continuously being written back to and read from registers R6 and R7. These values are marked in Figure 1.6 as “*”. Since they are produced and consumed only once, it is inefficient to store these values in the often limited register storage space. This situation, unfortunately, is inevitable when a vector operation, such as the one shown in Example 1.1, is expressed using a scalar program in a load-store ISA. □

A more efficient approach, is to chain a number of functional units together, with each unit performing a specific task. Thus when an intermediate value is produced by a functional unit, it is directly passed on to the next functional unit down the chain, thereby avoiding all the read and write traffic to the register file associated with this value.

Besides saving power, we could also pipeline the vector executions, such that one result could be produced in every cycle. The time required to perform a vector operation is given by Ts + n/Tr, where Ts is the initial setup cost, n is the vector length and Tr is the throughput rate in number of results produced per cycle. If a scalar machine takes m cycles to execute one iteration of the equivalent program loop, then the speedup using the vector machine is given by nm/(Ts+n/Tr), or nm/(Ts+n) for Tr=1. Maximum speedup could be achieved when Ts is sufficiently small and n is sufficiently large. In this case, the speedup approaches m, the number of cycles it takes for

the scalar machine to execute one iteration of the program loop. The relationship between speedup and n with Tr=1 is shown in Figure 1.7.

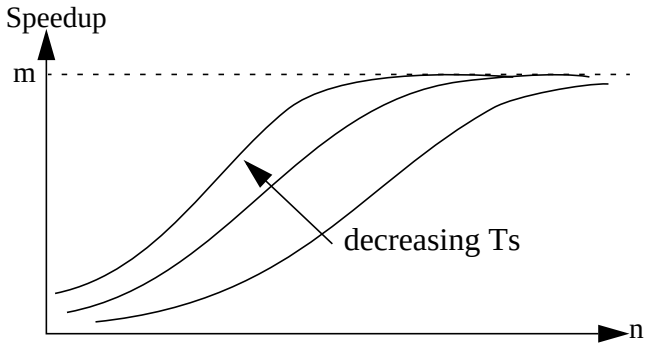


Figure 1.7: Relationships Between Speedup and n with Tr=1

Another subtle benefit of vector processing is its low instruction fetch bandwidth. Take Example 1.1 for example. The scalar program loop requires 8 instructions to be fetched during each iteration. In a vector machine, after fetching the initial vector setup code and the vector instruction itself, the machine does not need any further instruction fetch to perform the entire vector operation. As a result, the instruction fetch bandwidth and its associated memory traffic are much lower.

In summary, the strength of vector processing arises from:

- The ability to pipeline various operations on a single data stream (to improve performance);
- Efficient data storage and movement (large amount of temporary data are produced and consumed by adjacent functional units without going through the register file or the memory system); and
- Smaller routing area (result produced by a functional unit is routed directly to its destination functional unit, instead of broadcasting it to the entire datapath);
- Lower instruction fetch bandwidth.

Furthermore, efficient data movements and lower routing area could also mean lower power consumption. The strength of vector processing makes it very suitable for the low-cost, low-power embedded computing systems.

1.6 Vector Processing vs. Pseudo-Vector Processing

When a DSP algorithm or function is implemented on a DSP machine, it is often being transformed into program loops. The optimizing compiler then tries to re-structure the loop such that all the possible parallelism can be easily exploited by the machine.

Vector Processing

In mobile applications, some program loops are highly vectorizable, in the sense that they perform well defined vector operations. For example, a loop that performs a vector operation described by C[i] = A[i] + B[i], for all i, is a highly vectorizable loop. Highly vectorizable loops can be most efficiently executed using a "true" vector processing paradigm. In this paradigm, data are continuously streamed from the memory and are processed by a chain of functional units in a highly pipelined fashion. The processed data are then continuously streamed back to the memory. All temporary results produced during the vector operations are not written to or read from the register file.

Pseudo-Vector Processing

Some program loops, however, are impossible or difficult to vectorize. They may become vectorizable after being transformed by the compiler to "fit" the vector processing paradigm. These transformations involve adding some addition vector operations that may include mask generations, gather and scatter operations, etc. There are, however, overheads associated with these operations. In this dissertation, we will call these type of vector arithmetic *pseudo-vector arithmetic* (PVA).

For example, a loop that performs a vector operation described by C[i] = (A[i]>B[i])? $A[i]^2$: A[i]+B[i], for all i, is difficult (or costly) to vectorize. It is considered here to be a PVA arithmetic.

Many today's DSP machines can execute the PVA arithmetic efficiently. They include Analog Devices' ADSP-2106X SHARC™ chip [SHARC97] and Texas Instrument's TMS320Cxx™ family of chips [TMS320C3x]. These machines improve the performance by removing many of the overheads associated with: (i) loop control mechanism; (ii) constant-stride load; and (ii) constant-stride store.

1.7 The Basic Framework For This Dissertation

In this work, we will use the M-CORE instruction set architecture (ISA) for the purposes of illustration and evaluation. The M-CORE ISA uses 16-bit fixed length instruction encoding. It has one of the highest code density among all the commercially available ISA. Furthermore, the ISA provides extensive bits and bytes manipulation operations that are ideal for many real time embedded control and DSP applications [MCORE98,Moyer98].

An assembly program written for the pseudo-vector machine consists of regular scalar instructions (the M-CORE instructions in this case), with two additional vector instructions. The first vector instruction, the CVA instruction, vectorizes critical loops that are “highly vectorizable”. These vectorized loops are then executed on this machine using a “true” vector processing paradigm.

The second vector instruction, the PVA instruction, vectorizes critical loops that perform pseudo-vector arithmetic. These vectorized loops are then executed on this machine using a “pseudo”-vector processing paradigm.

For the purpose of this work, when a program loop is replaced by its equivalent code that consists of one or more vector instructions (CVA and/or PVA instructions), the program loop is said to be *vectorized*. This vectorized code segment is also called the *vector equivalent* of the original scalar program loop, as they both perform the same function. Vectorization can occur at the assembly level or at the source code level.

A program loop that can be vectorized only using CVA instruction(s) is called a *CVA vectorizable loop*. A program loop that can be vectorized using PVA instruction(s) is called a *PVA vectorizable loop*. The PVA execution represents a more general vectorizing mechanism. *Thus a loop that is CVA vectorizable, is also PVA vectorizable.*

Profile-Based Performance Evaluations

In this work, developing a vectorizing compiler for this machine is beyond the scope of this work. Without a vectorizing compiler, there is no vectorized assembly code. Without which, it is impossible to evaluate exactly the performance benefits by using a detail simulation model of the machine. Instead, the following approach is adopted.

The benchmarks were not re-compiled to vectorize the critical loops. Cycle-based simulations were first performed on a single-issued, four-stage pipelined machine. This scalar machine,

which executes M-CORE instructions, does not have any vector processing capability. The performance statistics collected on this scalar machine were used as a base result. This machine is also referred to here as the *base machine*.

The original scalar programs were dynamically profiled. Each program loop in these benchmarks was marked, and the number of invocations and the number of iterations were recorded. We then vectorized these critical loops by hand, at the assembly level.

The number of cycles saved for each loop were then computed using the profiled statistic. All the vector startup and exit costs were subtracted from these savings. We then summed up the net savings for each loop to give the total saving.

Throughout this dissertation, examples will be extensively used to illustrate some important concepts, procedures and techniques for vectorizing program loops. They will also be used to give detail illustrations on how we estimate execution cycle saving by using various vector constructs. These detailed examples may bore some of our readers; however, we think that this level of details is necessary in order to fatefully and accurately disseminate these information. The detail workings of evaluating the performance, for each critical loop, in each benchmark program, are given in Appendix B.

1.8 Profile-Based Performance Evaluations - An Example

We will use the following example to illustrate how we can estimate the performance improvements on this machine.

Example 1.2:

Vectorize the loop shown in Example 1.1 on page 9 using a CVA instruction. This loop is executed on a single-issued scalar machine for 100 iterations. This single-issued machine takes two cycles to execute a load instruction, and 2 cycles to execute a multiply instruction. Estimate the speedup when this loop is executed on the pseudo-vector machine using a CVA instruction.

This loop performs a vector operation described by C[i]=A[i]*B[i], for all i. The loop takes 12 cycles per iterations to execute, except for the last iteration where it takes only 11 cycles to execute (the branch instruction “bt” takes one cycle to execute when it is not taken; 2 cycles otherwise). Thus the execution cycles, on the scalar machine, is 12x99 +11x1 = 1199 cycles.

The loop can be vectorized using a CVA instruction as follows.

```
<Some initialization code>
// assign L0 to A, L1 to B, S to C.
CVA     mul  @L0, @L1, @S;
```

In this example, stream L_0, L_1 and S are enabled and are assigned to vector A, B and C, respectively. The CVA instruction multiplies, element-wise, between vectors A and B and writes the results to vector C.

The initialization code preceding the CVA instruction sets up the starting load/store addresses, operand sizes, constant stride values and vector length for vectors A, B and C. Executing this initialization code adds overhead to the vector executions. In addition, there are also other costs associated with this vector execution: vector instruction decode, pipeline warm-up cost, vector mode exit cost, etc. The total vector setup and exit costs, in this example, is 11 cycles (vector setup and exit costs will be described in detail in Section 6.4 in Chapter 6).

The P unit that performs the multiply function is fully pipelined. After these initial vector setup and exit costs, the pipeline can produce *one* result in *every* cycle. The execution cycles on the pseudo-vector machine is thus given by: 11 + 100 = 111 cycles. The speedup over the single-issued scalar machine is thus: 1199/111 = 10.80 . □

Unlike the traditional DSP processors, this machine can access three data streams per cycle during vector computations.

1.9 Contributions Of This Dissertation

As we will see later in this dissertation, with comparable hardware costs, for loops that are highly vectorizable, the CVA execution (the “true” vector processing) often offers higher performance benefits and lower power consumptions. For loops that are impossible or too costly to vectorize, the PVA execution (DSP’s style of processing) offers better performance benefits.

In this dissertation, we propose a CPU architecture to perform both the “true” vector arithmetic and the “pseudo” vector arithmetic on a single datapath. The optimizing compiler for this machine tries to “vectorize” the critical loops by selecting between a “true” vector processing paradigm, a “pseudo” vector processing paradigm, or a combination of both. As we will see later on, by providing the ability to execute both of these processing paradigms, we can achieve performance improvements that are higher than any of the individual paradigm.

A drawback of this dissertation is that a single-issued pipelined machine is used as a base machine. All performance results given in this dissertation are expressed in terms of improvements over this base machine. This dissertation would have been more interesting if the performance results are given relative to, say, a traditional DSP machine, similar to the ADSP-2106X SHARC chip.

Nevertheless, this dissertation has certainly explored some architectural *alternatives* to exploit ILP found in these embedded applications. In particular, it tries to exploit parallelism in a "vertical" (or "depth") direction, rather than in a "horizon" (or "width") direction as in a conventional wide-issued VLIW machine. This different will be described later in Section .

1.10 A Note On Vector Processing Paradigm

The first commercially available vector machine was the Cray-1^{TM} vector machine, built in 1974 [Cray1]. In this machine, multiple function units can be chained together to perform different scalar arithmetic functions simultaneously.

Later machines extended this idea to *array processing*, where a single instruction stream is executed by multiple PEs, each operates on a single data stream. These later machines can be classified as SIMD (single instruction stream and multiple data stream) vector machines. PowerPC's AltiVec^{TM} is a recent example of such machines [ALTIVEC98].

The Cray-1, on the other hand, has only one Processing Elements (PE). Each instruction in this machine only processes and consumes a single data stream. This machine can thus be classified as a SISD (single instruction stream and single data stream) vector machine.

Our pseudo-vector machine can also be classified as a SISD vector machine, since it has only a single PE, processing a single data stream. Figure 1.8 shows how the pseudo-vector machine relates to the various types of processing paradigms.

The rest of this dissertation is organized as follows. Chapter 2 describes related work. Chapter 3 classifies and discusses two types of vector arithmetic that can be performed on this machine. They are the CVA and the PVA arithmetic. Chapter 4 describes the programming model, on this machine, for vector executions. It also describes various special registers used for these executions. Chapter 5 describes the implementations of this machine. Chapter 6 describes the benchmarks used in this work and the performance evaluation methodologies. Vector setup costs and expressions for performance evaluation for typical program loops will also be derived in this

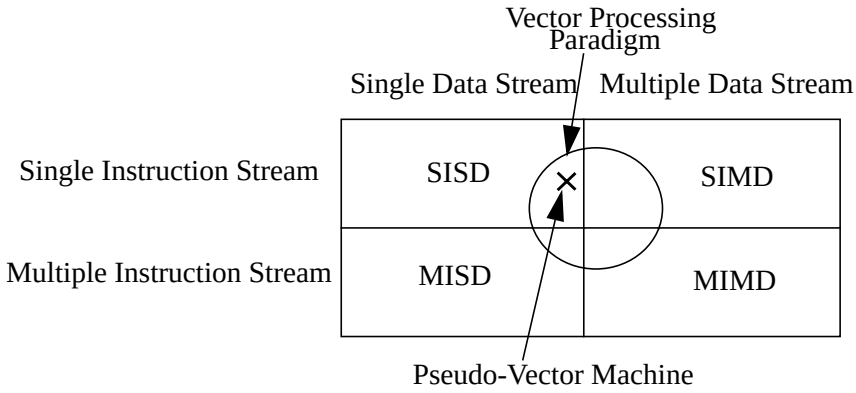


Figure 1.8: Various Processing Paradigms

Chapter. Chapter 7 presents some experimental results. Chapter 8 presents some architectural extensions for DSP applications. Chapter 9 summaries the dissertation.

CHAPTER 2

RELATED WORK

This Chapter describes related work to this dissertation.

2.1 Software Loop Unrolling

In software loop unrolling, multiple iterations from a loop are combined into a single iteration, with the branch instructions interleaving between the iterations being removed. The registers in each copy of the loop body are given different names to avoid unnecessary WAW (Write-After-Write) and WAR (Write-After-Read) data dependences. In a wide-issued machine (VLIW or superscalar machine), loop unrolling exposes the available ILP to the hardware by creating longer sequences of straight-line code.

Figure 2.1 shows an example of software loop unrolling. Figure 2.1(a) shows the original loop; Figure 2.1(b) shows that the loop is unrolled twice; and Figure 2.1(c) shows the unrolled loop is register renamed and re-scheduled so that dependent instructions are further apart. In the unrolled loops, R9 is assumed to be appropriately initialized for proper loop exit condition.

Software loop unrolling, however, results in a larger static program loop. Larger code size has an adverse affect on system cost as well as on instruction cache performance. In addition, software loop unrolling also increases register pressure on the compiler due to register renaming.

2.2 Software Pipelining and Register Rotation

Software pipelining is frequently used in wide-issued machines (VLIW and superscalar machines). This technique reorganizes loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. The scheduler

```
L1:
ld.h    r3,(r2)
add     r3, r10
st.h    r3, (r2)
subi    r2, 2
cmpnei r2,0
bt L1
```

(a) A loop that performs A[i]=A[i]+constant.

```
L1:
ld.h   r3,(r2)
add    r3,r10
st.h   r3,(r2)
subi   r2,2
ld.h   r3,(r2)
add    r3,r4
st.h   r3,(r2)
subi   r2,2
cmpnei r2,0
bt L1
```

(b) The loop is unrolled twice.

```
L1:
ld.h   r3,(r2)
ld.h   r7,2(r2)
add    r3,r10
add    r7,r10
st.h   r3,(r2)
st.h   r7,2(r2)
subi   r2,2
cmpnei r2,0
bt L1
```

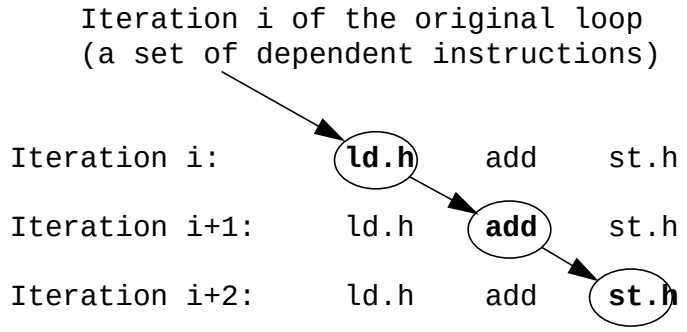
(c) The loop is register renamed and re-scheduled so that dependent instructions are further apart.

Figure 2.1: Software Loop Unrolling

essentially separates the dependent instructions that occur within a single loop iteration. Figure 2.2 shows conceptually how a loop is being software pipelined [Patterson96].

```
L1:
ld.h   r3,(r2)
add    r3, r10
st.h   r3, (r2)
subi   r2, 2
cmpnei r2, 0
bt L1
```

(a) A loop that performs A[i]=A[i]+constant.



(b) The loop is software pipelined

Figure 2.2: Software Pipelining

In a software pipelined loop, a multi-cycle instruction (such as a load instruction) may produce result that is only consumed in the next iteration; that is, an instruction may consume result that was produced in the previous iteration. To avoid over-writing a result before it is being consumed, the software pipelined loop may need to be unrolled and registers be renamed in software. Thus, software pipelining also enlarges code size and increases register pressure.

Register Rotation

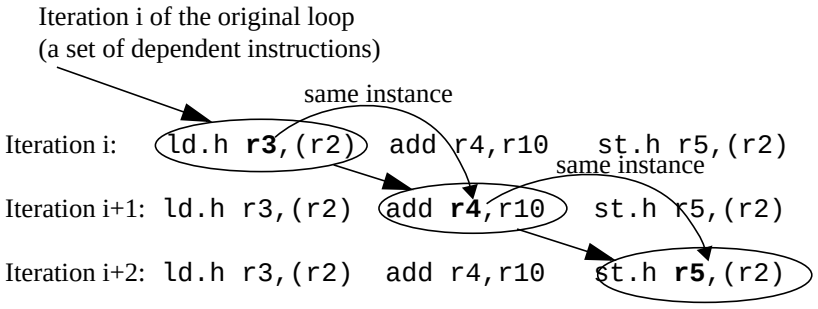
In Intel's IA-64^{TM} architecture, a technique called register rotation is used to minimize the code size expansion problem associated with software pipelining. In this technique, the hardware automatically renames the register by adding to the register number, a rotating register base (rrb) register. The rrb register is decremented when certain special software pipelined loop branches are

executed at the end of each iteration. Decrementing the rrb register makes the value in register X appear to move to register X+1 in the next iteration [IA64].

Figure 2.3 shows conceptually how a loop can be software pipelined using register rotation. In this example, register instance R3 in iteration i, is the same instance as register instance R4 in the iteration i+1; in turn, this register instance is the same instance as register instance R5 in iteration i+2, and so on.

```
L1:
ld.h   r3,(r2)
add    r3, r10
st.h   r3, (r2)
subi   r2, 2
cmpnei r2, 0
bt L1
```

(a) A loop that performs A[i]=A[i]+constant.



(b) The loop is software pipelined

Figure 2.3: Software Pipelining With Register Rotation

2.3 Stream Data Buffers In WM and SMC Architectures

In the WM architecture, Wulf first proposed using stream data buffers, under program control, to prefetch data from the memory into data buffers [Wulf92]. These buffers are organized as first-in-first-out (FIFO) queues. This technique can be illustrated using the following code segment for performing a vector dot product. In this code, “r0-r31” denotes integer registers and “f0-f31” denotes floating-point registers.

```
Sin32F       f0, r6, r5, 4
Sin32F       f1, r6, r5, 4

Loop:
f4 = (f0 * f1) + f4
JNI     f0, Loop
```

A single instruction, *streamin*, informs a stream control unit that a sequence of data operands is to be read from the memory, starting at a specified location, with a specified stride and count. The starting location and the count are specified in registers R6 and R5, respectively. The stride value (4 in this example) is specified in the `Sin32F` instructions (streamin 32-bit floating-point operands).

The first two `Sin32F` instructions set up two stream buffers. They designate registers f0 and f1 as the “head” of the two FIFO queues. Within the loop body, a read from one of these two registers dequeues a data item from the appropriate queue.

McKee et. al. [McKee95a,McKee95b] extended this work by proposing a Stream Memory Controller (SMC) architecture. In this architecture, multiple stream buffers were used to store: (i) data prefetched from the memory (input queue); and (ii) data to be stored to the memory (output queue). The heads of these FIFO queues appeared to the processor as memory mapped registers. After these streams are properly set up, a read from a designated memory location dequeues a data operand from an input queue; a write to a designated memory location enqueues a data operand to an output queue.

Figure 2.4 shows the block diagram of a SMC architecture. It shows how the CPU is interfaced with the memory system through a cache and a Stream Buffer Unit (SBU). The heads of the FIFO queues in the SBU appeared to the CPU as some pre-defined memory locations. These buffers are also used to buffer some single-use vectors to avoid polluting the cache.

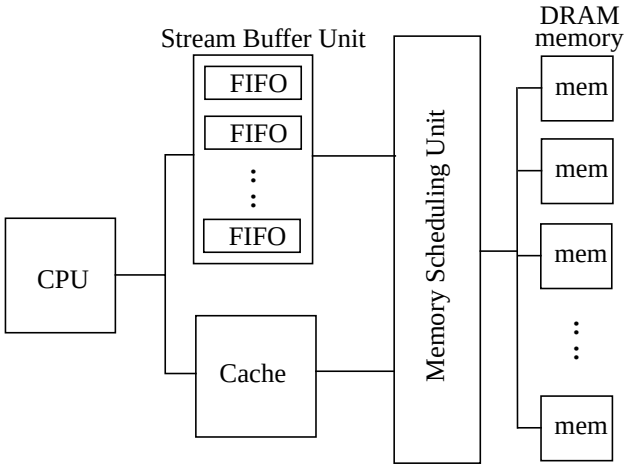


Figure 2.4: SMC Architecture - A Dynamic Access Ordering System

In addition to the stream buffers, there is also a Memory Scheduling Unit that dynamically reschedules the access requests made by the SBU and the cache. The unit coalesces and reschedules these requests to take advantage of the page access behavior of the DRAM memory [McKee95a,McKee95b].

2.4 Data Address Generators

On the Analog Devices SHARC ADSP-2106x CPU, there are two independent on-chip memory modules: the Program Memory (PM) and the Data Memory (DM). PM is used to store

program instructions and data; while DM is used to store data only. In this CPU, there are two Data Address Generation units, DAG1 and DAG2. These two DAGs can independently generate two load/store addresses, enabling the processor core to access two operands (any combinations of read and write) in each cycle. The block diagram of the ADSP-2106x CPU is shown in Figure 2.5 [SHARC97].

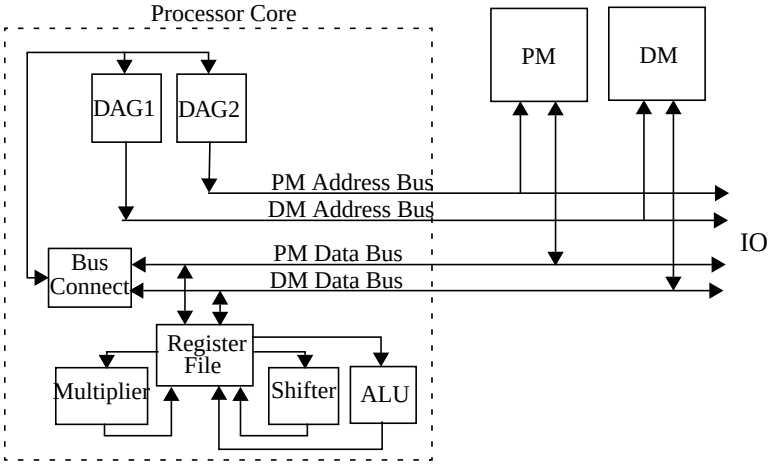


Figure 2.5: ADSP-2106x SHARC Block Diagram

Each of these DAGs contains eight I registers and eight M registers that are accessible to a program. A load or store instruction can access the memory by specifying an I register and a M register. For example, the following load instruction loads the memory content from the PM, with an address stored in I0; the value I0+M3 is then automatically written back to register I0 after the load operation (i.e. that address stored in I0 is automatically post-incremented by the amount stored in M3). The data loaded from PM is stored into register R6 [SHARC97].

```
R6 = PM(I0, M3); // indirect addressing with post-modify.
```

2.5 Compute And Memory Move Instructions

The ADSP-2106x chip can also provide a “compute-and-move” instruction that combines a compute function with up to two memory load/store operations. These operations are performed in parallel, in a single cycle. An example of such instructions is shown below.

```
R7 = R6 + R0,     DM(I0,M3)=R5,     PM(I11,M15)=R4;
```

This instruction performs an add, and two memory store operations, all in a single cycle. The two memory operations can be any combination of load and store operations.

The ADSP-2106x chip also provides some multifunction computations that combine parallel operations of the multiplier and the ALU, or the multiplier and dual functions in the ALU. The following shows two examples of such multifunction instructions [SHARC97].

```
R3 = R3 * R7,     R4 = R8 - R13;

R3 = R3 * R7,     R5 = R11 + R15,     R4 = R8 - R13;
```

There are, however, certain restrictions on how these independent operations can read their source registers. Some of these restrictions are illustrated in Figure 2.6. This Figure shows that each of the four input operands for computations that use both the ALU and multiplier are constrained to a different set of four register file locations. The X operand to the ALU, for example, can only be R8, R9, R10 or R11. This is a limited form of VLIW machine [SHARC97].

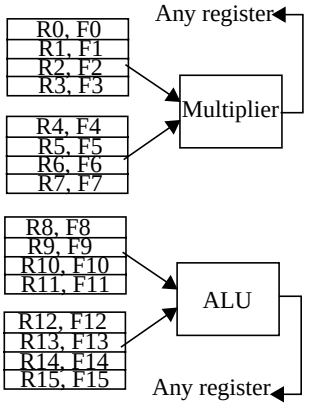


Figure 2.6: Source Registers For Multifunction Computations (ALU and Multiplier)

2.6 Special Loop Instructions For Removing Loop Control Overheads

Several commercial DSP ISA have special loop instructions for removing the overheads associated with the loop control mechanism [SHARC97,TMS320C3x,TRICORE97].

2.6.1 The TriCore™ ISA

In the TriCore ISA, for example, three special branch instructions are used to handle program loops. They are JNEI, JNED and LOOP instructions. The JNEI and JNED instructions are like normal jump-not-equal instructions, but with an additional increment or decrement operation on a data register operand. The LOOP instruction only requires execution time in the first and last iteration of the program loop. For all other iterations of the loop, the LOOP instruction has zero execution time. Here are some examples of loops using these instructions [TRICORE97].

```
mov d3, 3
L1:
. . . . .
jnei d3,10,L1
```

(a) A loop that executes d3=3,...,10

```
mov a2, 99
L1:
. . . . .
loop a2,L1
```

(b) A loop that executes 100 times

Figure 2.7: Program loops using special loop instructions

The JNEI, JNED and LOOP instructions in the TriCore ISA are capable of removing almost all the overheads associated with the branches at the end of the original loop. They do not, however, remove overheads associated with the cs-load and cs-store operations, if there is any in the loop.

2.6.2 The SHARC ADSP ISA

On the Analog Devices' ADSP-2106x chip, a DO UNTIL instruction is available for program loop executions. There are two types of DO UNITL loops. One is the counter-based and the other is not. In a counter-based loop, the iteration count is first written to a special register called Loop Count Register (LCNTR), prior to the loop execution. An example of such loop is shown below.

```
        LCNTR=30, DO label UNTIL LCE;
        . . . . loop body . . . .
label: [last instruction of the loop body]
```

The number of iterations can be specified as an immediate field in the DO UNTIL instruction; or, the instruction can also specify a universal register that contains the loop count. LCNTR is decremented by one for each iteration executed. The loop continues to execute until the Loop Counter Expires (LCE).

In a non-counter based DO UNTIL loop, the terminating condition is specified in the instruction. The iteration count, however, is not. In this case, the loop exits when the terminating condition is met. An example of such loops is shown below.

```
        DO label UNTIL AC; // exits when ALU Carry out is set
        . . . . loop body . . . .
label: [last instruction of the loop body]
```

This machine supports three hardware stacks: PC stack; Loop Address Stack and Loop Count Stack. These three stacks work in a synchronized manner for loop executions. When the ADSP-2106x executes a DO UNTIL instruction, the program sequencer pushes the address of the

last loop instruction and the termination condition for exiting the loop (both specified in the instruction) into the Loop Address Stack. It also pushes the top-of-loop address, which is the address of the instruction following the DO UNTIL instruction, on the PC stack. When a loop exits, all three stacks are popped. The three-stack mechanism allows the removal of loop control overheads for nested loops.

2.7 Vector Processing

A vector processor can be classified as a SISD machine, a SIMD machine, among other paradigms (see Section 1.10 on page 16).

2.7.1 Cray-1 Vector Machine - A SISD Vector Machine

The first commercially available vector machine, the Cray-1 vector machine, was primarily built for massively parallel scientific computations. Figure 2.8 shows the block diagram of the machine [Cray1].

There are three sets of primary registers: vector registers (V); scalar registers (S); and address registers (A). In addition, there are also scalar-save registers (T) and address-save registers (B). These registers act as buffers between the memory and the primary registers.

There are altogether 12 functions units in this machine. These functional units are organized into four groups: address, scalar, vector and floating-point pipelines. The vector pipelines obtain operands from one or two V registers and an S register. Results from a vector pipe are delivered to a V register. When a floating-point pipe is used for a vector operation, it can function similar to a vector pipe.

Once a vector operation is initiated, it will continue the operation until the number of operations performed equals a count specified by the Vector Length register. Vectors having a length greater than 64 are handled under program control in groups of 64 plus a remainder. This technique is also known as *strip mining*. A Vector Mask register was also provided to perform *masked vector operations*. In these operations, a vector operation on an element will only be performed if the corresponding bit in the Vector Mask register is set. These masked vector operations are used for vectorizing program loops with conditional branches and conditional executions within the loop body [Cray1,Hwang84,Hwang93].

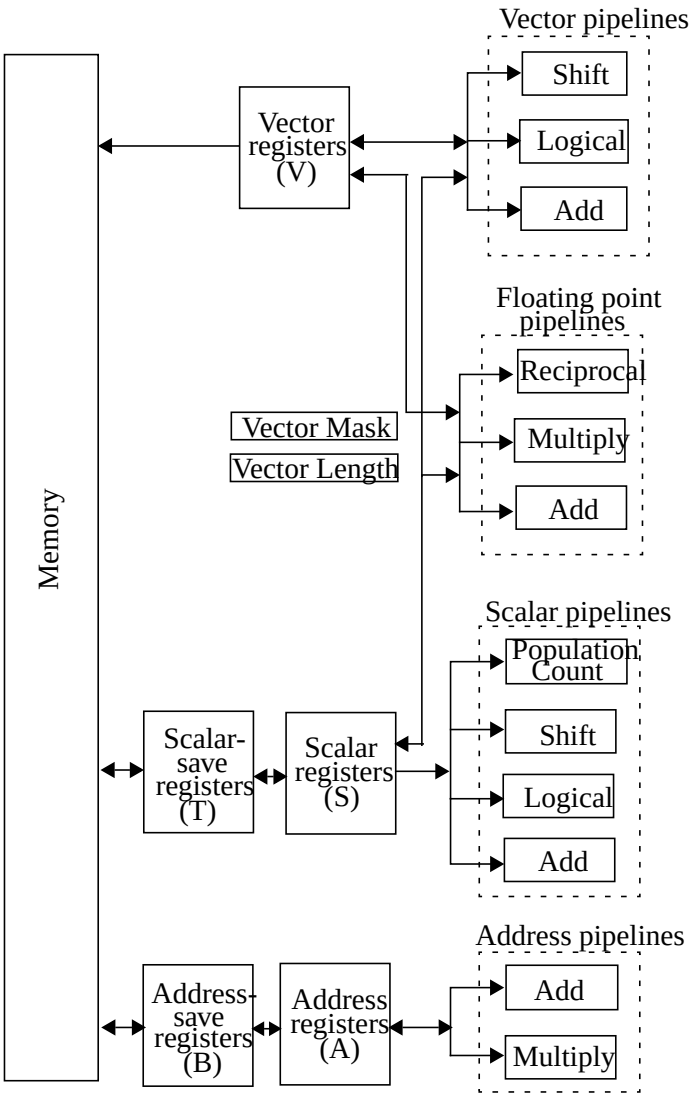


Figure 2.8: Block Diagram For Cray-1 Vector Machine

2.7.2 PowerPC AltiVec - A SIMD Vector Processor

The AltiVec technology provides a SIMD extension to the PowerPC architecture. A block diagram of PowerPC with AltiVec technology is shown in Figure 2.9 [ALTIVEC98].

Besides the usual Integer Unit and the Floating-Point Unit, it has a Vector unit which contains a 32-entry vector register file and a vector execution unit. The block diagram of the vector unit is shown in Figure 2.10. These register file and the execution unit are all fixed 128-bit wide. This width represents the total vector length, which can be subdivided into sixteen 8-bit bytes, eight 16-bit halfwords, or four 32-bit words. Depending on the operand type, the vector unit can simultaneously operate on 4 to 16 vector elements. This machine is essentially a multi-PE vector processor (or SIMD vector processor), although the number of PEs is a function of operand type.

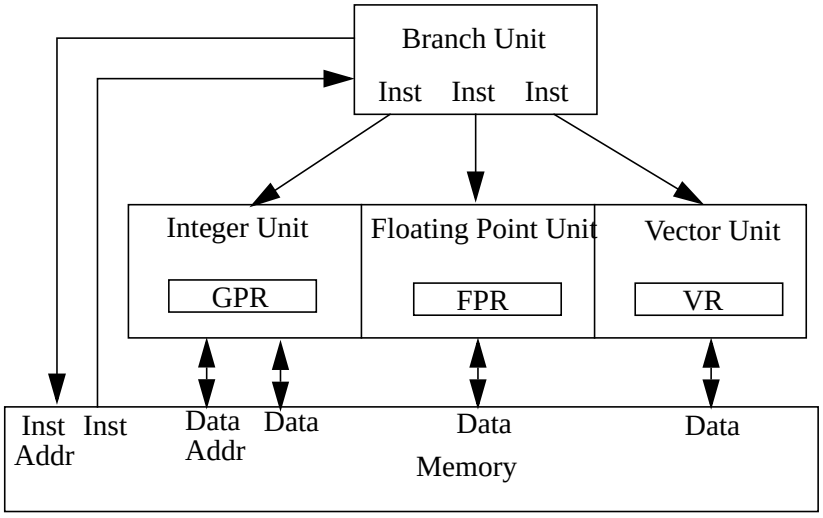


Figure 2.9: Block Diagram of PowerPC with AltiVec Technology

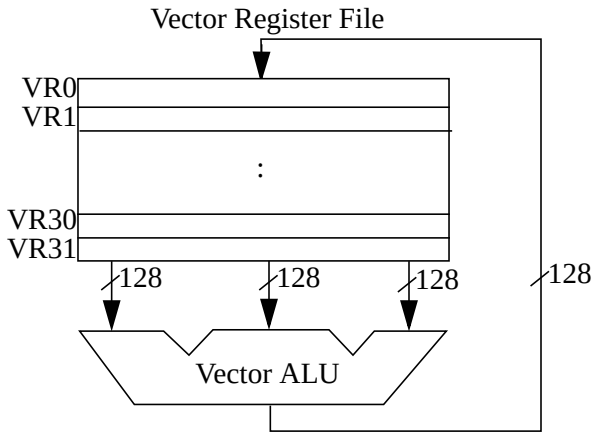


Figure 2.10: AltiVec Vector Unit

2.8 The MultiTitan Floating-Point Architecture: A Unified Vector/Scalar Floating-Point Architecture

Jouppi et. al. [Jouppi89] proposed a unified vector/scalar floating-pint architecture by adding a small amount of hardware to a scalar machine for improving the performance of classically vectorizable code.

In this architecture, a FPU unit (or FPU chip) is added to a CPU chip as a coprocessor. Both of these chips share the same off-chip data cache. All floating-point load/store operations are controlled by the CPU; the actual data transfers take place directly between the FPU chip and the data cache.

Figure 2.11 gives an overview of the architecture of the FPU. The FPU has three fully pipelined independent function units: add, multiply and reciprocal approximation. These functional units can accept a new set of operands in each cycle, and have a latency of three cycles.

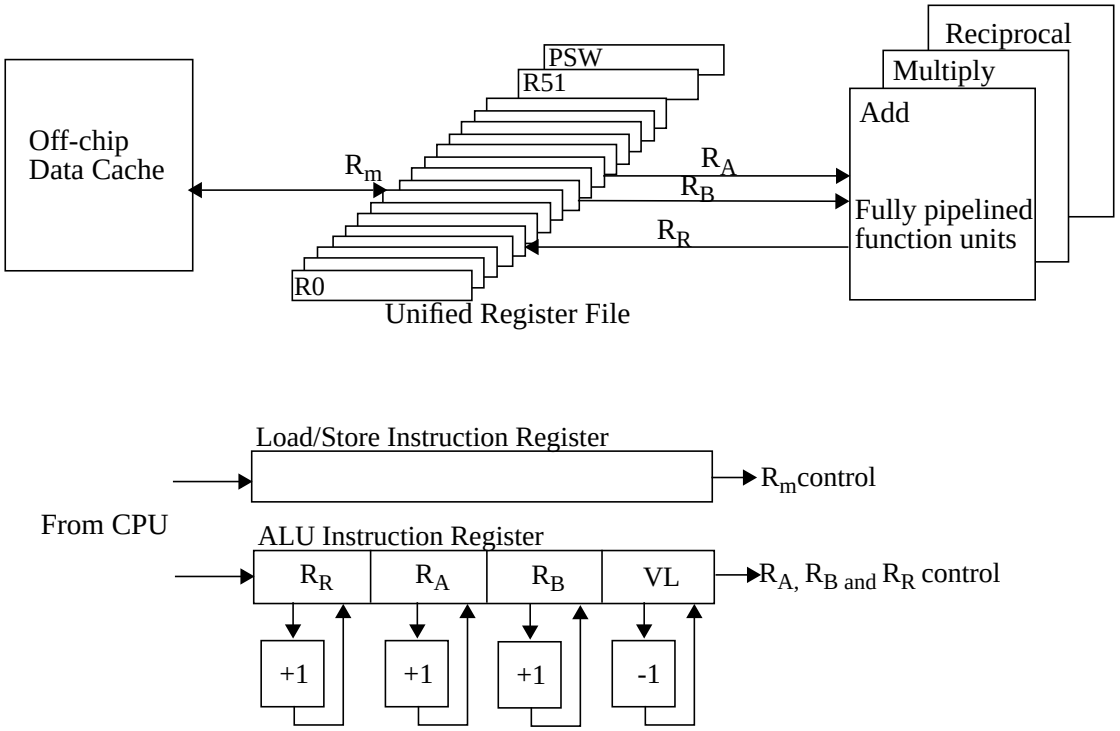


Figure 2.11: MultiTitan Floating-Point Architecture

A unified register file, containing 52 general-purpose 64-bit scalar registers, sits between the functional units and the data cache. This register file has four ports: two read ports for ALU source operands (R_A and R_B); one write port for ALU destination operand (R_R); and one read/write port for memory load/store operations (R_m).

There are two separate Instruction Registers: one controls the FPU ALU operations, the other controls the load store operations. These separate Instruction Registers allow the memory load/store operations to proceed in parallel with the issue of FPU ALU operations.

This architecture provides a single unified vector/scalar floating-point register file. Vectors are stored in successive registers in this register file. This allows individual vector elements to be addressed and accessed with scalar operations. Each arithmetic instruction contains a vector length field; scalar operations are simply vector operations of length one.

The format of a FPU ALU instruction is shown in Figure 2.12. The vector length (VL) field of an ALU instruction specifies the number of elements in the vector (between 1 to 16) to be pro-

cessed. When a vector instruction (VL field is non-zero) is loaded into the ALU Instruction Register, it remains in the register for the entire duration of this vector execution: each time a vector instruction is issued, the R_R, R_A and R_B fields of the Instruction Register are each incremented by one; and the VL field of the register is decremented by one. If the VL field is not zero, the vector instruction sitting in the Instruction Register is re-issued again; this process repeats until VL=0 (see Figure 2.11).

If the SR_A (or SR_B) bit in the instruction is set, register source field R_A (or R_B) does not increment (i.e. it is a scalar constant).

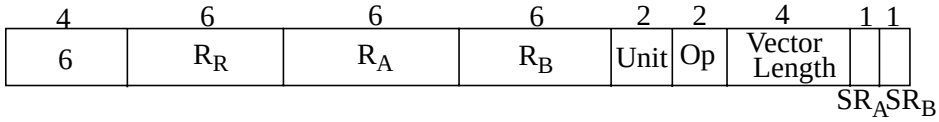


Figure 2.12: FPU ALU Instruction Format

In MultiTitan, the user can dynamically partition the 52 64-bit registers into any number of 1 to 16-element register groups (or vectors) on an instruction-by-instruction basis.

To facilitate concurrent load/store and ALU operations, the MultiTitan architecture also provides some hardware supports for synchronizing the reading and writing of individual register in the register file to avoid RAW, WAW and WAR hazards between these operations [Jouppi89].

2.9 Decoupled Access/Execute Machine - Astronautics ZS-1

The Decoupled Access/Execute Machine (Astronautics ZS-1) provides two loosely coupled datapath and two sets of registers: X registers (for floating point operands) and A registers (for integer operands). The block diagram of ZS-1 is shown in Figure 2.13.

The Instruction Fetch/Split unit fetches the instructions from the local memory and distributes them to two instruction pipelines: the X instruction pipeline and the A instruction pipeline. The fixed point and addressing instructions are forwarded to the A instruction pipeline; the floating point instructions are forwarded to the X instruction pipeline. Instructions distributed to these pipelines are decoded and issued to the appropriate functional units, if there is no hardware conflict nor data-dependency.

The ZS-1 uses two sets of FIFO (first-in-first-out) queues for communicating with the memory. One set consists of a A Load Queue (ALQ) and a A Store Queue (ASQ). These A queues are

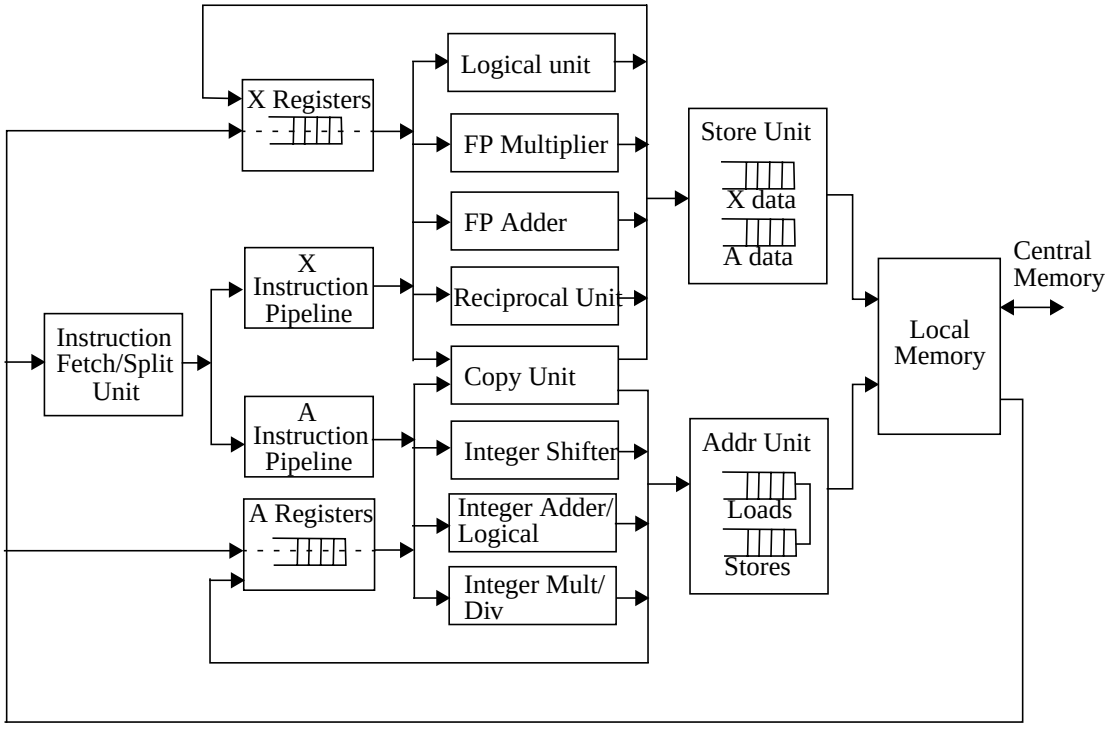


Figure 2.13: The Astronautics ZS-1

used in conjunction with the A registers. The other set of queues consists of a X Load Queue (XLQ) and a X Store Queue (XSQ). These X queues are used in conjunction with the X registers.

A unique feature of this architecture is that one of the instruction pipeline is allowed to run ahead of the other. These two pipelines “loosely” synchronize with each other using the four load/ store queues. This can be illustrated using the following example [Smith88,Smith89].

The ZS-1 loop consists of a mixture of integer/address instructions and floating point instructions. When the loop is fetched, these instructions are distributed appropriately to the A and X instruction pipelines. The first two instructions of the loop decrement the loop counter and test the result against zero. The next three instructions (executed on the A instruction pipeline) load elements of vector B, C and D. All three of the loaded data items will be placed in the X Load Queue (XLQ), as soon as they are available from the memory.

The next two instructions read B[i] into register X2, multiply it with C[i], and places the result into register X3. The next instruction adds D[i] to X3 and places the result to the head of the X Store Queue (XSQ). All these three instructions are executed on the X instruction pipeline; each of these instructions dequeues a data item from the XLQ queue.

By using these four load/store queues, memory load/store operations and the floating point computation can be performed concurrently. Furthermore, accesses to the prefetched data can be

```
          Do 10 I= 1,100
10        A(I) = B(I)*C(I) + D(I)
```

(a) FORTRAN Source Code

```
      A5 = -100        // negative loop count
      A6 = A-8         // pointer to A
      A7 = B-8         // pointer to B
      A8 = C-8         // pointer to C
      A9 = D-8         // pointer to D

loop: A5 = A5+1        // increment loop count
      B, A0=(A5==0)    // compare =0? set branch flag
      XLQ = (A7=A7+8)  // load next element of B
      XLQ = (A8=A8+8)  // load next element of C
      XLQ = (A9=A9+8)  // load next element of D
      X2 = XLQ         // copy B[i] into X2
      X3 = X2 * XLQ    // multiply B[i] and C[i]
      XSQ = XLQ + X3   // add D[i]; result to XSQ
      (A6=A6+8) = XSQ  // store result to A[i]
      JMPF loop        // branch on false to "loop"
```

(b) ZS-1 Assembly Code

Figure 2.14: A FORTRAN Loop and Its ZS-1 Assembly Code

achieved by reading from the heads of the appropriate load queues (XLQ or ALQ) [Smith88, Smith89].

2.10 The Transmeta's Crusoe™ Processors

Transmeta Corp. recently introduced a new family of processors called the Crusoe processors. These processors are x86-compatible. The x86 applications and the underlaying hardware are insulated with a software layer, called the Code Morphing™. The Code Morphine software dynamically "morphs" (or translate) the x86 code into the underlying hardware instructions. Figure 2.15 shows the applications, the Code Morphine software and the hardware layers of the machine. [CRUSOE00].

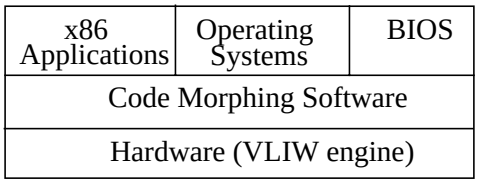


Figure 2.15: Applications/Code Morphing Software/Hardware Layers

The Code Morphine software can translate an entire group of x86 instructions at once, saving the resulting translation in a Translation Cache. When the x86 code is executed again, the translated code can be executed directly from the cache. Due to the locality of reference typically found in many application programs, a block of translated code is frequently re-used many times after it is being translated. As a result, the initial translation costs tend to be amortized over many executions.

There are a few advantages for insulating the legacy x86 code with the underlaying hardware. First, the hardware and software designers can judiciously rendered some functions in hardware and some in software, according to the product design goals and constraints. For a particular hardware implementation, only a special version of Code Morphine software needs to be ported.

Second, by having dynamic boundaries between the software and the hardware, larger design space can be explored and newer hardware/software techniques can be employed across different generations of the processor family, according to the technology available at a given point in time.

Third, compared to the conventional VLIW or superscalar machines which optimize the executions at the instruction level, the Code Morphine software has a higher level knowledge of the x86 code and can thus perform optimizations at a higher level.

In addition to dynamic translations, the Code Morphing software also instruments code into the translated code to dynamically “learn” about the program behaviors (such as block execution frequencies, and branch history). This data can be used later to decide when and how to spend the efforts for re-optimization [CRUSOE00].

2.11 Pseudo-Vector Machine - Comparisons With Related Work

The CPU architecture for our proposed pseudo-vector machine is shown in Figure 1.4 on page 8.

Comparison With The Crusoe Processors

The two initial Crusoe processors announced by Transmeta, model TM3120 and model TM5400, have a four-wide VLIW engine as their underlaying execution hardware. A unique feature of the Crusoe processors is the decoupling of the application software and the underlaying hardware implementation: instead of executing the target ISA (the x86 ISA) directly, the Code

Morphing software decomposes, optimizes and translates them into the native ISA (ISA executed on the native machine).

This dissertation work presents an architectural alternative for implementing the (underlaying) hardware, targeting mid-to-low end, lightweight, embedded mobile applications. It is possible to have an additional software layer, added to this machine, to insulate the hardware from the application software. In this case, each instruction on this machine can be thought of as a *macro-op* in the native ISA. That is, the "decoupling" approach used by the Crusoe processors is, in some sense, "orthogonal" to what is present here in this work.

Comparison With SISD Vector Machine, Cray-1

Unlike Cray-1, this pseudo-vector machine executes both the scalar and vector executions share the same datapath, using the same set of functional units. Furthermore, it does not have any fixed length vector register. Instead, it has a temporary vector memory (the TM module). This temporary storage space, to some degree, replaces the functionalities of vector registers.

One advantage of using TM over the vector registers is that it allows the compiler to have more flexibilities in organizing the temporary storage space - it can be organized to store longer but fewer temporary vectors, or shorter but more temporary vectors, or a mixture of vectors with different length, etc. This allows the compilers to select the TM's configuration for the best performance and lowest power consumption. TM and its associated vector allocations issues will be described and discussed in detail in Section 4.8.3 and Section 4.8.4 in Chapter 4.

Comparison With SIMD Vector Machine, PowerPC AltiVec

PowerPC AltiVec is a multiple PE vector machine, while the pseudo-vector machine is considered to be a single PE vector machine. The hardware cost of the latter is much lower than the former.

Comparison With The MultiTitan Unified Register File Architecture

The pseudo-vector machine shares a similar design goal with the MultiTitan architecture [Jouppi89]: they both attempt to improve the performance of vectorizable code by adding a minimum amount of hardware to a scalar machine.

Unlike the MultiTitan architecture, the pseudo-vector machine does not have a coprocessor. In particular, it only has one register file. In this machine, vectors are not stored in the register file; instead, they are streamed between the memory and the functional units directly. As a result, vec-

tor length is not limited to 16 elements; and strip-mining overheads for longer vectors are much lower or non-existence. Furthermore, limited storage space in the register file (reserved for frequently used scalar in this machine) is not clobbered by the vector data.

Comparisons With The VLIW Machines

The following distinctions can be made between the pseudo-vector machine and a VLIW machine, such as the ADSP-2106x chip.

- Unlike the ADSP-2106x chip, this pseudo-vector machine does not support multiple stacks (PC stack, loop address stack and loop count stack) for program loop executions. Without these stacks, only the inner most loop can benefit from eliminating the loop control overheads. Thus there will still be control overheads associated with the outer loops executions on this machine.

  In this machine, we concentrate our hardware resources on optimizing and speeding up the inner-most loop executions - since it provides the greatest benefits if the inner-most loop is executed multiple iterations each time it is invoked. If an inner loop executes *n* iterations, then saving one cycle from the inner-most loop executions will have the similar effect of saving *n* cycles from the next outer loop executions.

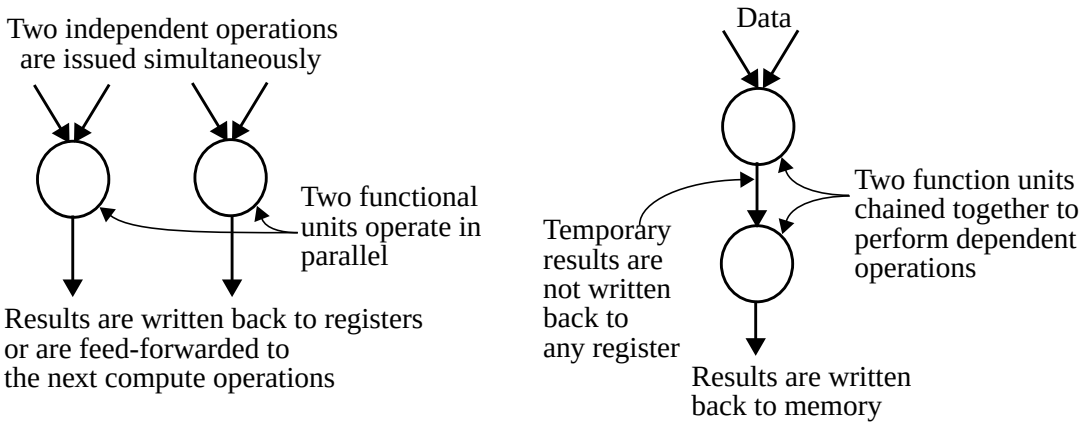
- In this machine, there are three independent on-chip memory modules, M0, M1 and TM. Thus the machine can perform up to three data memory operations (two reads and one write) per cycle.

- A VLIW machine is capable of issuing multiple independent compute operations in one cycle. This can be thought of performing multiple independent operations in a “horizontal” or “width” direction (see Figure 2.16). By the end of each cycle, the results produced by these operations are written back to some architectural registers; these results can also be feed-forwarded to other compute operations in the subsequent cycle.

  In the pseudo-vector machine, when executing in a CVA mode (or a “true” vector mode), two function units are chained together in a “vertical” direction to perform two dependent operations. These operations are performed simultaneously on two different data. The temporary results that are produced between the two functional units are not written back to any architectural register.

  The differences between the execution model for a two-wide VLIW machine and those for a

two-deep vector machine can be illustrated in Figure 2.16.



(a) A two-wide VLIW machine (b) A two-deep vector machine

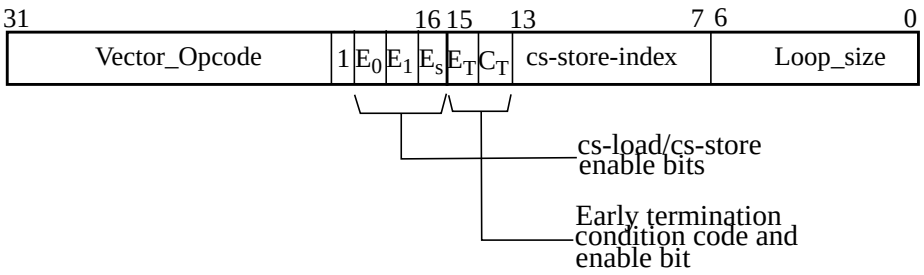
Figure 2.16: Comparison Between a 2-Wide VLIW Machine and a 2-Deep Vector Machine

- When executing in a PVA mode, the execution model of the pseudo-vector machine is very similar to those of a conventional DSP processor (such as the ADSP-2106x Chip). However, the following minor differences can be noted.

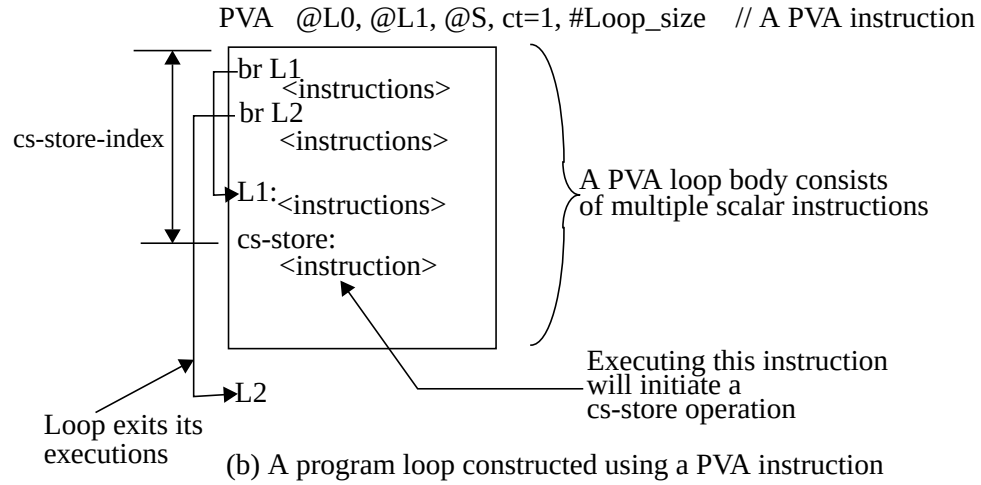
    - Each program loop on this machine can optionally enable up to two cs-load (the L_0 and L_1 streams) and one cs-store (the S stream) operations. The instruction format of a PVA instruction and the general structure of a program loop constructed using a PVA instruction are shown in Figure 2.17.

    - In the PVA instruction, $E_0/E_1/E_S$ are the enable bits for the $L_0/L_1/S$ streams, respectively. Access to the L_0 and L_1 streams within the loop body is achieved by reading from registers R0 and R1, respectively. In addition, a label within the loop body, called “cs-store”, marks the instruction that, when executed, will automatically initiate a cs-store operation. The data for the store operation, is the same data written back by that instruction.

    - There are three ways in which a loop can exit its executions: (i) by the Count Register becoming zero; or (ii) by the conditional code becoming a value pre-specified in the C_T field (assuming E_T=1); or (iii) by a taken branch within the loop body with its target lies outside the loop body. All these three exit mechanisms can co-exist simultaneously on a program loop.



(a) PVA Instruction Format



(b) A program loop constructed using a PVA instruction

Figure 2.17: The General Structure of a PVA Program Loop

2.12 General Comparisons With Related Work

The pseudo-vector machine has merged many architectural techniques found in conventional DSP and vector machines, into a single execution model. It performs two forms of vector arithmetic (namely, the “true” and “pseudo” vector arithmetic) on a single datapath. At certain time, this machine behaves like a true vector machine; and in some other time, it behaves like a DSP machine that optimizes program loop executions.

The optimizing compiler, in this case, decides which processing paradigm is best suited for a given program loop. In general, it will first try to vectorize a loop using CVA instruction(s), in an attempt to exploit the low-power and high performance aspects of vector processing paradigm. If this is not possible, or too costly, it will then try to vectorize the loop using a PVA instruction (the DSP’ style of loop-based executions), or a combination of both CVA and PVA instructions.

If a vector arithmetic can be described by one or more CVA depicted in Figure 1.5 on page 9, then CVA executions offer great opportunities for performance improvements. This is particularly true when the p_op is a multi-cycle operation, and can be fully pipelined at the P unit.

CHAPTER 3

VECTOR ARITHMETIC

The pseudo-vector machine is capable of executing in scalar mode and in vector mode. When in a vector mode, all the vector arithmetic that can be performed on this machine can be categorized into the following two categories.

- Canonical Vector Arithmetic (CVA);
- Pseudo-Vector Arithmetic (PVA).

Correspondingly, there are two vector instructions in this machine, one for CVA executions and one for PVA executions. In this Chapter, we will in turn describe each of these two types of vector arithmetic. In addition, there are also some vector arithmetic that can terminate its execution before the full vector is being processed. We will call this a *vector arithmetic with early termination*.

3.1 Canonical Vector Arithmetic

Many vector arithmetic can be represented by a generic data dependency graph shown in Figure 3.1. In this graph, "•" denotes some scalar arithmetic or logical function. This dependency graph is *generic*, in the sense that it represents a large number of vector operations. The vector arithmetic shown in Example 1.1 on page 9, for example, can be represented by the dependency graph shown in Figure 3.1. The scalar function, in this case, is the multiplication function.

We will now further generalize this dependency graph as follows.

- A source operand can be either a scalar constant or a vector element;
- The generic dependency graph shown in Figure 3.1 is chained with another dependency graph to form a compounded operation with three source operands and two arithmetic functions (see Figure 3.2(a)).

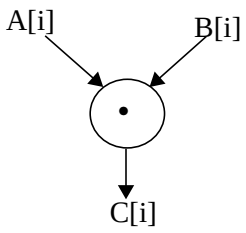


Figure 3.1: A Generic Data Dependency Graph

- A feed-back path is added from the output to the input of the second arithmetic function (see Figure 3.2(b) and Figure 3.2(b)).

We will call these generalized vector operations *canonical vector operations*, or *canonical vector arithmetic* (CVA)*. Figure 3.2 shows three basic types of CVA. They are: (i) *compound CVA*; (ii) *reduction CVA*. and (iii) *hybrid CVA*. In this figure, "s" denotes some scalars quantities.

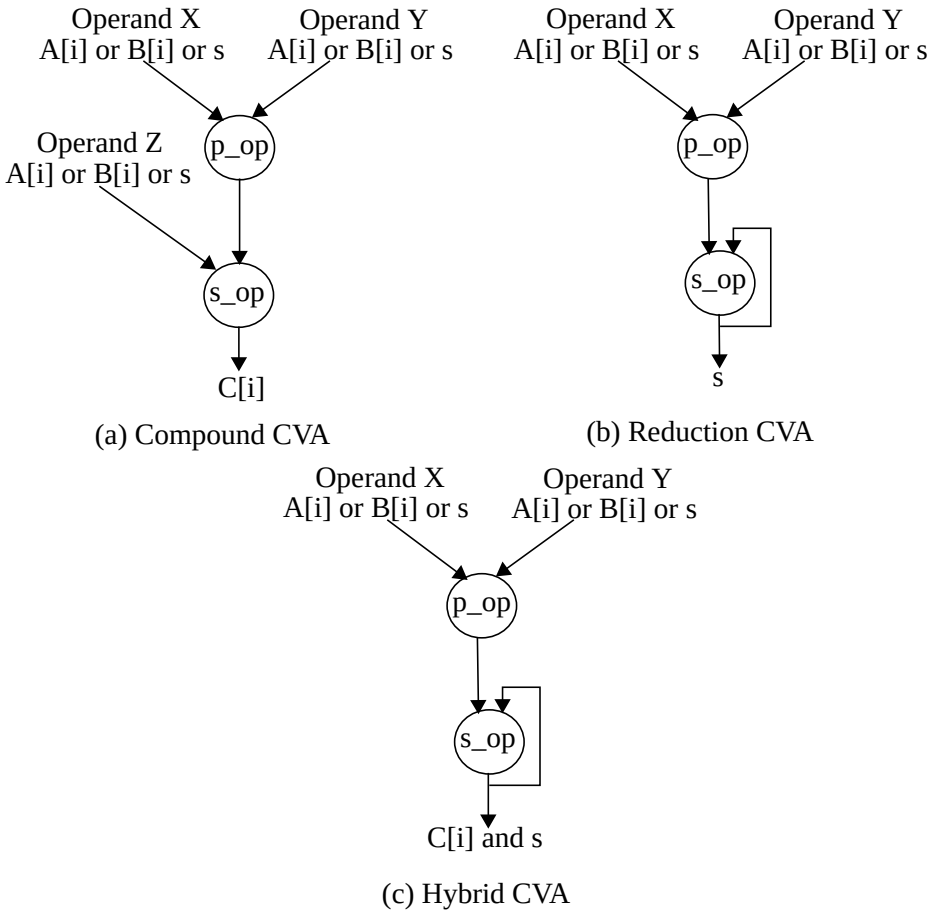


Figure 3.2: Dependency Graphs for Three Types of CVA

---

* In this dissertation, vector arithmetic and vector operation will be used interchangeably.

In these three types of CVA, the first arithmetic performed near the two inputs is called the *primary arithmetic* (p_op). The second arithmetic performed near the output is called the *secondary arithmetic* (s_op).

In order to limit the hardware costs, for the purpose of this work, we will impose the following restrictions.

- In the compound CVA, the two arithmetic combined can only source up to two distinct vectors (vectors A and B in Figure 3.2(a)). This restriction, however, does not preclude the case where all three source operands are vectors. It reduces the maximum memory bandwidth requirement, from three data fetches per cycle to two data fetches per cycle.
- The secondary arithmetic, s_op, is limited to a few simple commutative ALU functions. These include add, and, or, xor, etc.*

The basic datapath structure for executing the CVA is shown in Figure 3.3.

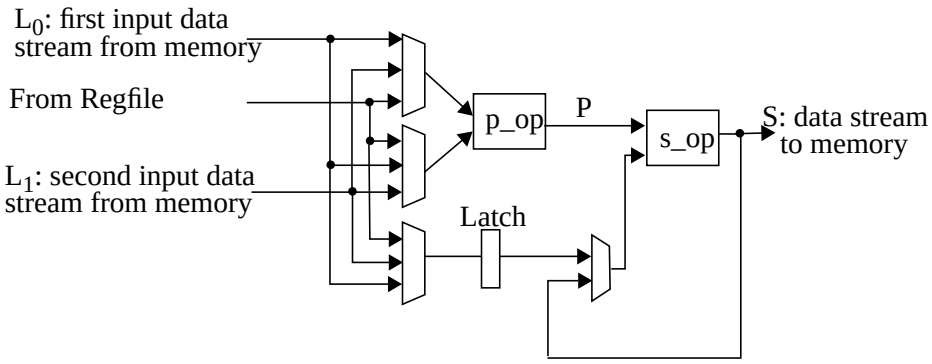


Figure 3.3: Basic Datapath Structure for Executing CVA

3.1.1 Compound CVA

Compound CVA can produce a destination vector as a result of vector computations. The general form of a compound CVA can be expressed as follows.

- If source X and source Y are all vectors, then

  $R_i = (X_i \text{ p\_op } Y_i)$ s_op Z_i, i=0,...,n-1

where

n denotes the vector length;

p_op denotes the primary arithmetic;

---

* A scalar arithmetic, *op*, is said to be *commutative* if (x *op* y) = (y *op* x), for all scalar x and y.

s_op denotes the secondary arithmetic;

R_i denotes the *i*th element of the destination vector;

X_i, Y_i and Z_i are respectively the *i*th element of vectors X, Y and Z.

- If source X is a scalar constant, x, and source Y is a vector, then

  $R_i = (x \text{ p\_op } Y_i) \text{ s\_op } Z_i$, i=0,...,n-1

- If source X is a vector and source Y is a scalar constant, y, then

  $R_i = (X_i \text{ p\_op } y) \text{ s\_op } Z_i$, i=0,...,n-1

The secondary arithmetic, s_op, can also be a “no-op”. In this case, the dependency graph shown in Figure 3.2(a) degenerates into those shown in Figure 3.1.

3.1.2 Reduction CVA

A reduction CVA performs a *vector reduction operation*, where one or more vectors, as a result of the vector operation, is reduced to a scalar result [Hwang84]. The general form of a reduction CVA can be expressed as follows.

- If source X and source Y are all vectors, then

  $S_0 = (X_0 \text{ p\_op } Y_0)$
  $S_i = (X_i \text{ p\_op } Y_i) \text{ s\_op } S_{i-1}$, i=1,...,n-1;
  $R = S_{n-1}$

where

S_i denotes the *i*th partial result;
R denotes the scalar result for the vector reduction operations.

- If source X is a scalar constant, x, and source Y is a vector, then

  $S_0 = (x \text{ p\_op } Y_0)$
  $S_i = (x \text{ p\_op } Y_i) \text{ s\_op } S_{i-1}$, i=1,...,n-1;
  $R = S_{n-1}$

- If source X is a vector and source Y is a scalar constant, y, then

  $S_0 = (X_0 \text{ p\_op } y)$
  $S_i = (X_i \text{ p\_op } y) \text{ s\_op } S_{i-1}$, i=1, n-1;
  $R = S_{n-1}$

The feed-back path shown in Figure 3.3, in conjunction with the secondary arithmetic, are responsible for computing and accumulating a *partial result*; and eventually, producing a final scalar result. An example of such reduction operations is the inner product of two vectors, described by $\sum_i$(A[i] * B[i]). In this case, the primary arithmetic is the "multiplication" function and the secondary arithmetic is the accumulative "add" function.

3.1.3 Hybrid CVA

A hybrid CVA is identical to a reduction CVA, except that the partial results are also constantly being written to a destination vector. The general form of a hybrid CVA is identical to those for reduction CVA, except that the partial results, S_i, i=0,.,n-1, also form a destination vector, R, with $R_i = S_i$, i=0,.,n-1. For hybrid CVA, there are two destinations: a scalar destination and a vector destination.

3.1.4 Some Examples Of CVA

A few examples of CVA are shown in Table 3.1. In each of these examples, the corresponding CVA vector instruction is also shown in Table 3.1.

In these CVA instructions, "@" denotes a data stream. In particular, "@L0" denotes the first input data stream from the memory; "@L1" denotes the second input data stream from the memory; "@P" denotes the intermediate result stream produced by the primary arithmetic, p_op; and "@S" denotes the output data stream to the memory (compare these with Figure 3.3 on page 40). All L_0, L_1 and S streams are constant stride memory operations.

For compound CVA, the CVA instructions can specify both the primary and the secondary arithmetic (see Example (a) and Example (b) in Table 3.1). These two arithmetic are specified in the CVA instructions with a comma separating them: the primary arithmetic is specified first, followed by the secondary arithmetic. The instruction is terminated with the ";" symbol. In this case, the "@P" stream appears as a destination in the primary arithmetic; it also appears in the second arithmetic as a source.

For compound CVA, the CVA instructions can also specify only the primary arithmetic (see Example (c) through Example (f) in Table 3.1). The secondary arithmetic, in this case, is a "no-op" and the results produced by the primary arithmetic are stored directly to the memory via "@S" stream. No "@P" stream is specified in the instructions.

Table 3.1: Some Examples of CVA

Ex.	Vector Arithmetic	Descriptions	CVA Vector Instructions	Streams Enabled		
				L_0	L_1	S
(i) Compound CVA						
a	C[i]= sA[i] + B[i]	Vector constant multiplication and addition	CVA mul r4, @L0, @P, add @P, @L1, @S;	Y	Y	Y
b	C[i]= $(A[i])^2$+B[i]	Element-wise square and add	CVA mul @L0, @L0, @P, add @P, @L1, @S;	Y	Y	Y
c	C[i]= $(A[i])^2$	Element-wise square	CVA mul @L0, @L0, @S;	Y	N	Y
d	C[i]= abs(A[i])	Element-wise absolute	CVA abs @L0, @S;	Y	N	Y
e	C[i]=A[i]	Vector assignment	CVA mov @L0, @S;	Y	N	Y
f	C[i] = 0	Memory block initialization	CVA mov 0, @S;	N	N	Y
(ii) Reduction CVA						
g	IP = $\sum_{i=0,n-1}$ (A[i] * B[i])	Vector inner product	CVA mul @L0, @L1, @P, add r3, @P, r3;	Y	Y	N
h	$Norm^2$ = $\sum_{i=0,n-1}$ $(A[i])^2$	The square of "norm" of vector A	CVA mul @L0, @L0, @P, add r3, @P, r3;	Y	N	N
i	Sum = $\sum_{i=0,n-1}$ A[i]	Vector reduction through summations	CVA mov @L0, @P, add r3, @P, r3;	Y	N	N
(iii) Hybrid CVA						
j	C[i]=A[i]*B[i]; IP = $\sum_{i=0,n-1}$ (A[i] * B[i])	Vector multiplication and vector inner product	CVA mul @L0, @L1, @P, add r3, @P, {@S,r3};	Y	Y	Y

For reduction CVA, the CVA instructions specify both the primary and secondary arithmetic (see Example (g) through Example (i) in Table 3.1). In these cases, the "@P" stream appears as a designation in the primary arithmetic; it also appears in the secondary arithmetic as one of the source operands. The destination and the second source operand of the secondary arithmetic is always register R3. For reduction CVA, R3 is designated to store the partial results as well as the final scalar result for the reduction operations.

Since the secondary arithmetic is commutative, a shorthand notation can be used to describe a reduction CVA. In this case, the entire secondary arithmetic expression is replaced by the function name for s_op. The CVA instruction for calculating the inner product (Example (g) in Table 3.1), for example, can also be written as:

```
CVA mul @L0, @L1, add; // shorthand notation for reduction CVA
```

For hybrid CVA, the instruction syntax is similar to those for reduction CVA, except that the secondary arithmetic has two destinations: a S stream and register R3. They appear on the CVA instruction in the form “{@S, R3}”. There is no shorthand notation for hybrid CVA.

3.2 Pseudo-Vector Arithmetic

There are many program loops that perform arithmetic functions on vectors, but cannot be classified as vector arithmetic. Consider the following program loop.

Example 3.1:

```
for (i=0; i<n; i++) {
       if (A[j]>m) {
            “M”
       }else{
            “N”
      }
       if (B[k]>A[j]) goto EXIT;
       j=j+4; k=k+8;
}
EXIT
```

In this loop, “M” and “N” represent some segments of straight-line code. They are conditionally executed based on condition A[j]>m. Also, there are more than one possible exits from this loop. The conditional executions of “M” and “N” and conditional loop exits can base on some runtime information. The control flow graph of this loop is shown in Figure 3.4. □

Often there is no systematic way of describing these types of operations in vector form. When implemented in assembly code, this loop is converted into a program loop composed of assembly instructions, with multiple conditional and unconditional branches in between them. We will call such an arithmetic a *pseudo-vector arithmetic* (PVA).

In a “true” vector machine (such as Cray-1), the loop shown in Example 3.1 can still be vectorized using multiple vector operations. To simplify our discussion, let’s assume that the statement “if (B[k]>A[j])” is removed from the loop. In this case, the condition (A[j]>m) can be used to generate a vector mask. Assuming that “M” and “N” represent some vector operations. Then “M” can be performed under the control of the mask; while “N” can be performed under the con-

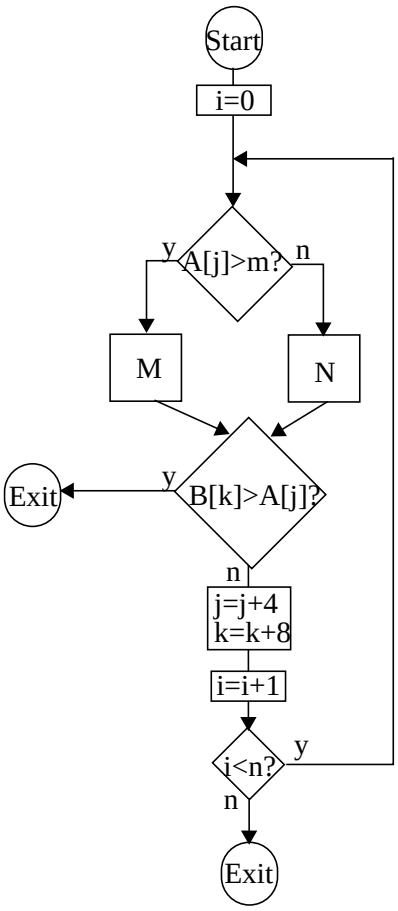


Figure 3.4: A Program Loop with Multiple Exits

trol of a “inverted” version of the mask. In this example, no gather nor scatter operation is needed.

In our pseudo-vector machine, the PVA execution is a generalization of CVA execution. *Any program loop that is vectorizable using the CVA construct, is also vectorizable using the PVA construct. The converse, however, is not true*. If a loop is vectorizable by using either a CVA or a PVA construct, it is usually more efficient, in terms of execution time, to vectorize it using the CVA construct. There are also some program loops that are best vectorized by a combinations of CVA and PVA constructs. We will come back to this subject later in this dissertation.

Example 3.2:

Vectorize the vector operations: $C[i]= A[i]^2+B[i]$.

The CVA version of this program loop shown in Example (b) in Table 3.1. It can also be vectorized, using a PVA construct, as follows.

```
<Some initialization code>
```

```
// assign L0 to A, L1 to B, S to C.
PVA  @L0, @L1, @S, #3;      // PVA instruction
mov   r3, r0                // r3 = A[i]
mul   r3, r3                // r3 = A[i]^2
cs-store:
add   r3, r1                // C[i] = A[i]^2 + B[i];
                            // initiate a cs-store here
```

In this vectorized loop, all L_0, L_1 and S streams are enabled. In particular, the L_0 stream is assigned to vector A; the L_1 stream is assigned to vector B and the S stream is assigned to vector C. “Assignment” here refers to initializing some specially designated registers to the appropriate load/store addresses and stride values for accessing the vectors A, B and C.

Within the loop body, access to A[i] is achieved by reading from register R0; access to B[i] is achieved by reading from register R1. R0 and R1 are, respectively, the “heads” of the L_0 and L_1 streams.

An “add” instruction is located at a label called “cs-store” within the loop body. When this instruction is executed, a constant-stride store to the memory is automatically initiated, using the result written back by this instruction. This cs-store operation writes a data element to the destination vector C.

The size of the PVA loop, in this example, is three instructions (“mov”, “mul” and “add”). This is specified in the PVA instruction using the notation “#3”. □

3.3 Vector Arithmetic with Early Termination

Consider the program loop shown in Example 3.3.

Example 3.3:

```
L1:
ld.h    r7,(r2)       // load A[i]
addi    r2,2
ld.h    r6,(r3)       // load B[i]
addi    r3,2
cmplt   r6,r7         // is A[i] > B[i]?
bt     EXIT           // if so, exit the loop
decne  r14            // if not, decrement the count
```

```
bt     L1              //is the entire vector being processed?
                       //if not, branch backward
EXIT:
```

The corresponding high level source code is shown below.

```
for (i=0; i<n; i++) {
    if (A[i] > B[i]) {break;}
}
```

This loop performs an element-wise compare between vector A and B. This loop exits as soon as A[i]>B[i]. If no such pair of elements exists, then all the elements of vector A and B will be processed before the loop exits. □

If a program loop performs certain arithmetic function on a fixed length vector(s), and it is possible for the computation to terminate even before the last element(s) of the vector(s) is being processed, then such an operation is called a *vector arithmetic with early termination*. Example 3.1 and Example 3.3 shown earlier in this Chapter are examples of such arithmetic.

In a vector arithmetic with early termination, there are two terminating conditions. One is when all the elements of a source vector are being processed. The other is when certain arithmetic condition is met. This condition could be met prior to the last vector element is being processed. This condition is usually data dependent and can not be determined a priori.

When a loop with “if” condition in the loop body is vectorized on a “true” vector machine, it is often converted into multiple vector operations which involves mask generation, and possibly some gather and scatter operations.

Early Termination Enable Bit, E_T

In our vector execution model, both CVA and PVA arithmetic described in Section 3.1 and Section 3.2 can be a vector arithmetic with early termination. All vector executions can terminate early by setting an enable bit, called the *Early Termination Enable Bit* (E_T bit), in the vector instruction. The value of the condition code for this to occur is also specified in the instruction. This bit is called the C_T bit. The formats of the CVA and PVA instructions will be described in Section 4.4 in Chapter 4.

For CVA executions with early termination enabled (E_T=1), the primary arithmetic are some functions that can alter the condition code. During the course of the vector executions, if the con-

dition code is set to the pre-specified value (given by the C_T bit), the vector execution will terminate immediately.

Example 3.4:

Vectorize the program loop shown in Example 3.3 using a CVA construct.

This loop can be vectorized using a CVA construct as follows.

```
<Some initialization code>
// assign L0 to A, L1 to B.
CVA     cmplt.ct=1   @L0, @L1;
```

In this CVA instruction, the secondary arithmetic is unspecified (i.e. it is a “no-op”). In this instruction, E_T=1 and C_T=1. The primary arithmetic (“cmplt”) compares, element-wise, between vector A and vector B. If A[i] > B[i], for some i, the condition code is set to one, terminating the CVA executions. If no such pair of elements are found, executions continue until the two vectors are exhausted. □

For PVA executions, *early termination* can be achieved by any of the following two mechanisms:

- By using the E_T and C_T bits in the PVA instruction, similar to those described above; and
- When a branch within the loop body is taken, and the target of the branch lies outside the loop body.

When any of the above two conditions is met, the PVA executions terminate automatically. The machine will then enter a scalar mode. These two exit mechanisms are illustrated in the following example.

Example 3.5:

Vectorize the program loop shown in Example 3.3 using a PVA construct.

The CVA version of this loop is shown in Example 3.4. This loop can also be vectorized using a PVA construct as follows.

```
<Some initialization code>
// assign L0 to A, L1 to B.
PVA     @L0, @L1, ct=1, #1;
cmplt   R0, R1                  // loop body
```

Like its CVA counterpart, the PVA instruction has E_T=1 and C_T=1. The PVA loop body consists of only one scalar instruction (“cmplt”). This instruction compares, element-wise, between vectors A and B. This is achieved by reading and comparing R0 and R1. When the condition code c is set to 1 (C_T=1) as a result of the comparisons, the PVA executions terminate immediately.

An alternate version of this loop is shown below.

```
<Some initialization code>
// assign L0 to A, L1 to B.
PVA     @L0, @L1, #2;
cmplt   R0, R1                  // part of loop body
bt EXIT                         // part of loop body

EXIT
```

In this alternative, E_T=0 in the PVA instruction. There are two instructions in the loop body: (“cmplt” and “bt”). When the second instruction in the loop (“bt EXIT”) is taken (the target of this branch lies outside the loop body), the PVA executions terminate immediately. Otherwise, the executions will continue with the first instruction of the next iteration (“cmplt”).

The first PVA version of the loop executes as fast as the CVA version shown in Example 3.4, taking one cycle per iteration to execute. The second PVA version is less efficient, taking two cycles per iteration to execute. By setting E_T=1, we eliminated the overhead associated with executing the conditional branch instruction. □

The “exit-by-conditional-branches” alternative is typically used by a program loop with conditional executions within the loop body that also utilize the condition code. An example of such loops is shown in Example 3.1 on page 44.

CHAPTER 4

PROGRAMMING MODELS

In this Chapter, we will discuss the programming models for vector executions on this machine.

In the CVA executions, there are notions of “vector elements” and “vector length”. In the PVA executions, there are notions of “loop body” and “iterations”. Throughout this dissertation, “vector length” and “number of iteration” will be used interchangeably, if such uses do not cause any confusion in a given context.

4.1 Execution Modes

There are two major execution modes in this machine: scalar mode and vector mode. The vector mode can be further divided into: CVA mode and PVA mode. Correspondingly, there are two vector instructions, a CVA instruction and a PVA instruction. Furthermore, there are three types of CVA executions. Figure 4.1 shows all the execution modes in this machine.

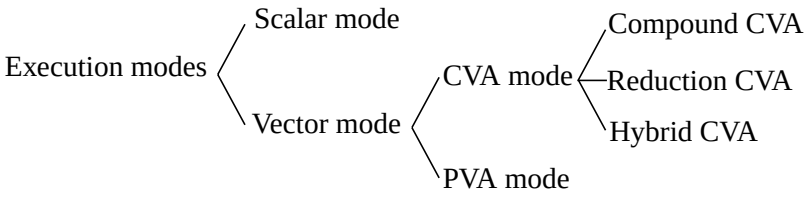


Figure 4.1: Various Execution Modes On The Pseudo-Vector Machine

4.2 Constant-Stride Load/Store Operations

A vector instruction (a CVA or a PVA instruction) can optionally enable up to two constant-stride load (cs-load) and one constant-stride store (cs-store) operations to be performed for each pair of vector elements (for CVA executions) or during loop executions (for PVA executions). The

two cs-load operations are denoted by L_0 and L_1. The cs-store operations are denoted by S. L_0 and L_1 are also referred to as the input data streams; S is also referred to as the output data stream.

The data dependency graphs in Figure 4.2 show the relationships between L_0, L_1 and source operands X, Y and Z. Operand X can source either from L_0, L_1 or register R4. Operand Y can source either from L_0, L_1 or register R5. Operand Z can source from either L_0, L_1 or register R6.

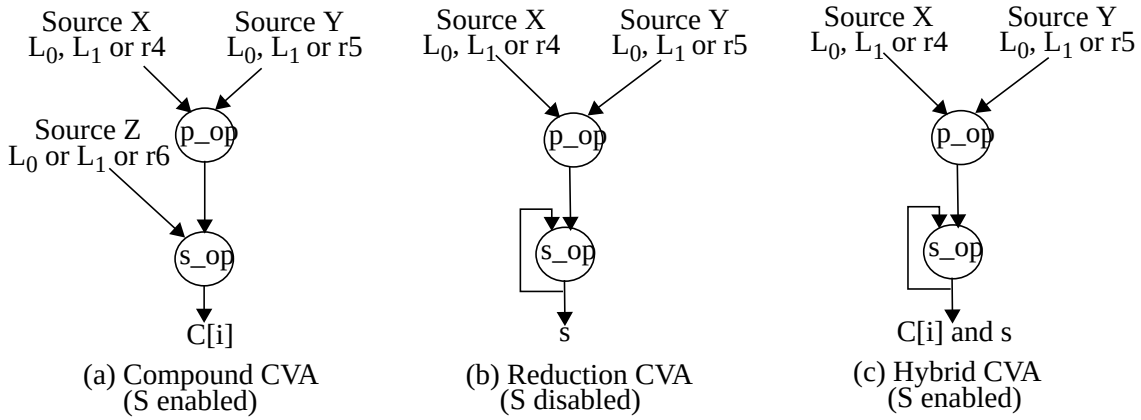


Figure 4.2: Dependency Graphs Showing the Relationships Between L_0, L_1 and Source Operands X, Y and Z

4.2.1 cs-load And cs-store For CVA Executions

When in a CVA mode, S can be enabled or disabled. For compound CVA, S is always enabled. If S is disabled, the CVA corresponds to a reduction CVA. If S is enabled, the CVA corresponds to either a compound CVA or a hybrid CVA.

Example 4.1:

Vectorize the following program loop.

```
L1:
stw         r5,(r14)
addi        r14,4
decne       r6
bt          L1
```

This loop is found in a benchmark program called `summin`. It initializes a block of memory with a constant. It can be vectorized using a CVA instruction as follows.

```
<Some initialization code>
CVA     mov r4, @S;
```

This is a compound CVA. In this case, S is enabled but L_0 and L_1 are disabled. The initialization code preceding the CVA instruction sets up the starting store address, its operand size and constant-stride value. R4, in this case, is initialized to the appropriate constant prior to the CVA executions. The cs-store S are then performed using this constant. The primary arithmetic, in this case, is a "mov" function. Besides setting up the store address and stride value, the vector initialization code also includes the initialization of vector length to a special register called *Count Index Register* (CIR). This register, which controls the number of vector elements to be processed (for CVA executions) or the number of iterations to be executed (for PVA executions), will be described further in Section 4.3.2. □

Example 4.2:

Vectorize the following program loop.

```
L1:
ld.b        r7,(r3)
st.b        r7,(r2)
sub         r2, r9
sub         r3, r9
decne       r6
bt          L1
```

This loop performs a block memory transfer between two memory locations. R9, in this case, contains the stride value for the cs-load and cs-store operations. This loop can be vectorized as follows.

```
<Some initialization code>
CVA       mov @L0,@S;
```

This is also a compound CVA. In this case, L_0 and S are enabled, but not L_1. The primary arithmetic is also a "mov" (or "pass") function. The secondary arithmetic is also a "pass" function. □

Example 4.3:

Vectorize the following vector operation: C[i] = sA[i] + B[i], for some scalar constant s.

This vector operation can be vectorized using a CVA instruction as follows.

```
<Some initialization code>
// assign L0 to A, L1 to B.
```

```
// initialize r5 with the scalar s
CVA     mul @L0, r5, @P,     add @P, @L1, @S;
```

This is another compound CVA. In this example, all L_0, L_1 and S streams are enabled. Stream L_0 is assigned to vector A; stream L_1 is assigned to vector B and stream S is assigned to vector C. Register R5 is initialized to the scalar s prior to the vector executions. The primary arithmetic is the “mul” function and the secondary arithmetic is the “add” function. □

In the above Example, the CVA instruction sources two vector operands. A CVA instruction can also source up to three vectors, provided that the three source vectors are actually coming from no more than two distinct vectors. For example, the vector operations $C[i] = A[i]^2 + B[i]$ can be vectorized as follows.

```
// assign L0 to A, L1 to B, S to C.
CVA   mul @L0, @L0, @P,    add @P, @L1, @S;
```

In this case, operands X, Y and Z are all vectors, sourcing from only two distinct vectors, A and B.

Example 4.4:

Vectorize the following program loop.

```
L1:
ldw      r10,(r14)
decne    r4
mov      r7,r10
lsr      r7,r9
or       r7,r3
mov      r3,r10
stw      r7,(r13)
lsl      r3,r8
addi     r14,4
addi     r13,4
bt       L1
```

This loop is taken from a benchmark program called `blit`. The vector operations performed by this loop can be described by:

```
C[i] = (lsr(A[i], r9)   |   lsl(A[i], r8))
```

This loops reads in a vector, A, an element at a time, and performs a logical shift right (“lsr”) and a logical shift left (“lsl”) operations on the element. It then “or” the results of these two shift operations and writes it to a destination vector, C. It can be vectorized using two CVA instructions, as follows.

```
<Some initialization code>
// assign L0 to vector A, S to a temporary vector.
mov r5, r9
CVA     lsr @L0, r5, @S;
<Some initialization code>
// assign L0 to the temporary vector, L1 to vector A.
// assign S to vector C.
mov r5, r8
CVA     lsl @L0, r5, @P,     or @L1, @P, @S;
```

Both of these CVA instructions are compound CVA instructions. The first performs a “lsr” operation on the source vector A and produces a temporary vector. The second CVA instruction has “lsl” as its primary arithmetic and “or” as its secondary arithmetic. This latter instruction reads in the temporary vector via L_0 and performs a “lsl” operation on it. It also reads the original source vector A via L_1 and performs a “or” function with the results produced from the primary arithmetic. It then writes back the results to vector C via S.

Notice that the source operand Y for both CVA instructions is always sourced from register R5. Additional “mov” instructions are thus needed to initialize this register prior to the CVA executions. The data dependency graph for this loop is shown in Figure 4.3.

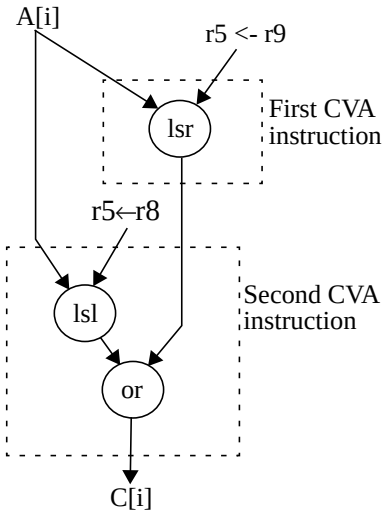


Figure 4.3: Data Dependency Graph for Example 4.4

Note that in the original scalar program loop shown on page 53, there are 11 instructions in the loop body. Thus during *each* iteration, there were 11 instruction requests being made to the instruction memory. In the CVA executions, besides fetching the initialization codes and the "mov" instructions, there is no instruction request throughout the course of vector executions. □

Example 4.5:

Vectorize the following program loop.

```
movi     r4,1
L1:
stw      r4,(r11)    // cs-store
addi     r11,4
addi     r4,1        // store data sequence: 1,2,3,...
cmplt    r4,r13      // check for loop terminating condition
bt       L1
```

This loop is taken from a benchmark program called `auto`. The loop stores a sequence of data described by: 1,2,3,..., etc. This loop can be vectorized as follows.

```
<Some initialization code>
movi    r4,1
CVA     mov r4, @P,     add @P, r3, {r3,@S};
```

This is a hybrid CVA. The partial result, generated in every cycle, is simultaneously written to register R3 and stream S.

If the starting value of the sequence is some constant other than a one (say, the constant is s), then two CVA instructions will be needed. The first one is the hybrid CVA described above. The second is a compound CVA that adds s-1 to each element of the resulting vector produced by the first CVA executions. □

4.2.2 cs-load And cs-store For PVA Executions

When in a PVA mode, any or all of L_0, L_1 and S can be disabled. To enable the cs-load L_0 (or L_1), the "@L0" (or "@L1") symbol must appear in the PVA instruction. When L_0 (or L_1) is enabled, access to the cs-load data stream within the loop body is achieved by reading from register R0 (or R1). Each read from R0 (or R1) dequeues one data item from the L_0 (or L_1) stream. Registers R0 and R1, however, are read-accessible only. Writing to these registers will be ignored by the hardware.

To enable cs-store S, there must be one (and only one) “cs-store” label within the loop body. When the instruction located at this label is executed, a cs-store operation is automatically initiated, using the data written back by that instruction. There can be no more than one “cs-store” label in a PVA loop body. If there is none, S is considered to be disabled by the assembler.

Example 4.6:

Vectorize the following program loop.

```
for (i=0; i<n; i++) {
      if (A[i] > p) C[i]=4 else C[i]=8;
}
```

The corresponding assembly code is shown below.

```
mov         r4, 4
mov         r8, 8
L1:
ld.w        r7,(r3)   // load A[i]
addi        r3,4
mov         r2, r8
cmplt       r6,r7     // is A[i] > p?
movt        r2, r4    // r2 = (A[i] > p)? r4: r8;
st.w        r2, (r4)  // store result
addi        r4,4
decne       r5        // decrement the loop count
bt          L1        //is the entire vector being processed?
```

This loop can be vectorized using a PVA construct as follows.

```
<Some initialization code>
// assign L0 to A, S to C.
mov r4, 4
mov r8, 8
PVA       @L0, @S, #4;
mov       r2, r8
complt    R6, R0          // Is (A[j] > p)?
cs-store:
movt      r2,r4           // r2 = (A[i] > p)? r4: r8;
                          //cs-store performed here
```

In this loop, L_0 and S are enabled, but not L_1. In each PVA iteration, a cs-store operation is

automatically initiated whenever the “movt” instruction is executed.

Note that after vectorizing this loop with a PVA instruction, the size of the loop body is reduced from the original 9 instructions, down to 3 instructions. The instruction request bandwidth when executing this loop is thus greatly reduced.

All the instructions that were eliminated from the loop performed some repetitive loop control or cs-load/cs-store operations. During loop executions, the machine no longer needs these instructions since their associated operations are already pre-specified in the PVA instructions, or in some special registers. □

4.3 Special Registers For Vector Executions

Prior to any vector execution, certain registers need to be properly initialized. These special registers contain all the necessary information for the hardware to carry out the proper vector executions. Table 4.1 shows these special registers. They are described in the followings.

Table 4.1: Special Registers For Vector Executions

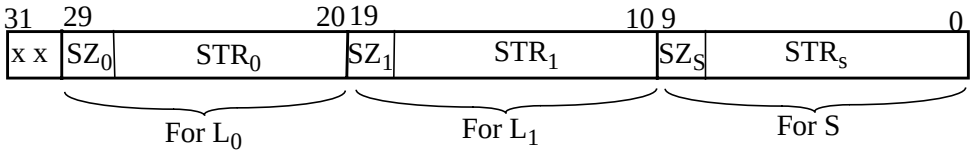
Registers	Notations	Register Contents
Stride and Size Register	SSR	Stride values and operand sizes for L_0, L_1 and S
Count Index Register	CIR	Number of iterations to be executed
General purpose register, R0[a]	R0	Load address for L_0
General purpose register, R1[a]	R1	Load address for L_1
General purpose register, R2	R2	Store address for S
General purpose register, R3	R3	Partial and final results for reduction or hybrid CVA
General purpose register, R4	R4	Optional source for operand X
General purpose register, R5	R5	Optional source for operand Y
General purpose register, R6	R6	Optional source for operand Z

a. These are overlaid instances. See Section 4.6.

4.3.1 Stride and Size Register

A special register, called the *Stride and Size Register* (SSR), is used to specify the stride values and the operand sizes for L_0, L_1 and S streams, if the corresponding load/store operation is enabled. This register is partitioned into three independent fields, one for each of the L_0, L_1 and S

stream. The format of SSR is shown in Figure 4.4.



$STR_0/STR_1/STR_S$: Stride value for L_0, L_1 and S, respectively.

$SZ_0/SZ_1/SZ_S$: Operand size for L_0, L_1 and S, respectively.
They are defined as follows:

0x	word
10	halfword
11	byte

Figure 4.4: Stride Size Register, SSR

The $STR_0/STR_1/STR_S$ fields specify the stride values for L_0, L_1 and S, respectively; the $SZ_0/SZ_1/SZ_S$ fields specify the operand sizes for L_0, L_1 and S, respectively.

When in a PVA mode, if any of L_0, L_1 and S is disabled, its corresponding “SZ” and “STR” fields are “don’t care”.

SSR is a special control register that can be accessed using the “move-to-control register” instruction (“mtcr”) or the “move-from-control register” instruction (“mfcr”).

4.3.2 Count Index Register (CIR)

A special register, called the *Count Index Register* (CIR), is composed of two independent registers: (i) the *Count Register* (CR); and (ii) the *Index Register* (IXR). This register is shown in Figure 4.5.

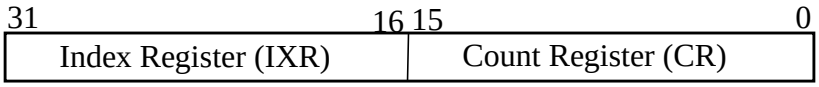


Figure 4.5: Count Index Register, CIR

Register CR is used to specify the vector length for CVA executions, or the number of loop iterations to be executed for PVA executions. This register is initialized by the software prior to its vector executions. During the vector executions, this register is automatically decremented by one, for each vector element (or each iteration) being processed. When this register reaches zero, the vector executions will terminate. A vector arithmetic, however, could also terminate prior to CR becoming zero. This will be described further in Section 4.5. CR is a read and write accessible register.

Register IXR is only used for PVA executions. It serves as a “local PC” within a PVA loop body. When we are executing the first instruction in the loop body, the IXR is set to one. When we are executing the *i*th instruction in the loop body, the IXR is set to *i*. IXR is a hardware-only register. The software have no access to this register. Reading from this register returns all zeros. Writing to this register is ignored by the hardware.*

Register CIR is a special control register that can be accessed using the “move-to-control” instruction (“mtcr”) or the “move-from-control” instruction (“mfcr”). However, only the lower portion of CIR (the CR register) is accessible to the software.

4.3.3 Register For Storing Constant-Stride Load Addresses

The load addresses for L_0 and L_1, if enabled, are stored in registers R0 and R1, respectively, as shown in Table 4.1. Prior to entering a vector execution, the software initializes these registers to point to the starting load addresses for L_0 and L_1. During the course of vector executions, these registers are constantly updated by the hardware, to point to the latest cs-load addresses. That is, each time a cs-load L_0 (or L_1) is performed, R0 (or R1) is added by the stride value specified in the STR_0 (or STR_1) field in the SSR register. The hardware, in this case, updates the overlaid instances of R0 and R1. Register Overlay will be described in Section 4.6.

4.3.4 Register For Storing Constant-Stride Store Addresses

If S is enabled, the store addresses for S are stored in register R2. Similar to R0 and R1, this register is constantly updated by the hardware to point to the latest cs-store address. R2, however, is not overlaid. Again, Register Overlay will be described in Section 4.6.

4.3.5 Scalar Results For Reduction And Hybrid CVA

When executing a reduction CVA or a hybrid CVA, R3 is designated as the destination register. It is also designated for storing the partial results during the vector executions. That is, the partial results produced during the course of the vector executions are constantly being written back to register R3.

4.3.6 Scalar Source Operands For CVA Executions

During CVA executions, operands X, Y and Z can optionally source from registers R4, R5 and R6, respectively. Once initialized by the software, these registers will not be altered during the entire course of CVA executions.

---

* The hardware recovery mechanisms for exceptions and interrupts for PVA executions will be discussed in Section 4.7.

When in PVA executions, registers R3, R4 and R5 have no special meaning and are for general uses.

4.4 Vector Instructions

Figure 4.6 shows the formats of the CVA and PVA instructions. Both the CVA and PVA instructions are 32-bit wide.

In both CVA and PVA instructions, the E_0, E_1, E_s bits, respectively, enables or disables the L_0, L_1 and S streams. Also in these instructions, the E_T bit enables or disables the early termination capability. The C_T bit specifies the condition code for this to occur, if the capability is enabled.

In the CVA instruction, the V_{x0}/V_{x1}, V_{y0}/V_{y1} and V_{z0}/V_{z1} bits appropriately select the sourcing of operands X, Y and Z. In particular, these bits select the sourcing of L_0, L_1 or a designated registers, as defined in Figure 4.2 on page 51. The p_op field in the CVA instruction specifies a primary arithmetic. The s_op field specifies the secondary arithmetic.

In the PVA instruction, the cs-store-index field specifies the index of the instruction in the loop body that will initiate a cs-store operation, when the instruction is executed. The first instruction in a PVA loop has an instruction index of zero, the second instruction in the PVA loop has an instruction index of one, and so on. The Loop_size field in this instruction specifies the size of the PVA loop body, in number of scalar instructions.

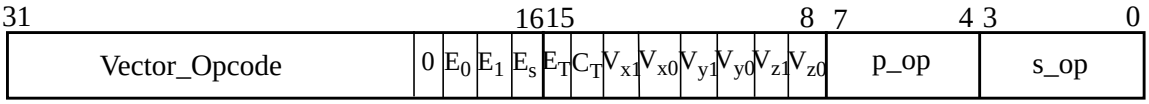
Example 4.7:

Implement the inner product of two vectors: $\sum_{i = 0, n-1}$(A[i]*B[i]).

```
Initialize CR to n.
Initialize SSR[STR0] to stride value for vector A.
Initialize SSR[STR1] to stride value for vector B.
Initialize R0 to the starting address for vector A.
Initialize R1 to the starting address for vector B.
CVA     mul @L0, @L1, add;
```

In this example, S is disabled. Thus this is a reduction CVA. The primary arithmetic for this reduction operations is the “mul” function; the secondary arithmetic is the “add” function. When the vector computation is completed, the final result (the inner product) will be implicitly written

(a) CVA Instruction



Vector_Opcode: Opcode for vector instructions

$E_0/E_1/E_S$: Enable bits for data streams L_0, L_1 and S, respectively

V_{x0}/V_{x1}: Defined as follows.

$V_{x1}V_{x0} = 00$ operand X sources from R3
$V_{x1}V_{x0} = 01$ operand X sources from L_0
$V_{x1}V_{x0} = 10$ operand X sources from L_1
$V_{x1}V_{x0} = 11$ operand X sources zero

V_{y0}/V_{y1}: Defined as follows.

$V_{y1}V_{y0} = 00$ operand Y sources from R4
$V_{y1}V_{y0} = 01$ operand Y sources from L_0
$V_{y1}V_{y0} = 10$ operand Y sources from L_1
$V_{y1}V_{y0} = 11$ operand Y sources zero

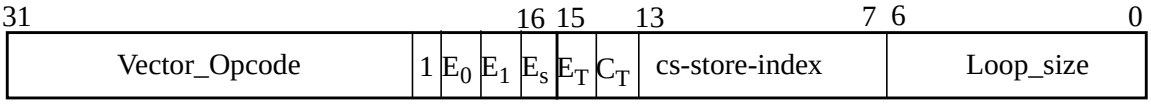
V_{z0}/V_{z1}: Defined as follows.

$V_{z1}V_{z0} = 00$ operand Z sources from R5 (compound CVA)
$V_{z1}V_{z0} = 01$ operand Z sources from L_0 (compound CVA)
$V_{z1}V_{z0} = 10$ operand Z sources from L_1(compound CVA)
$V_{z1}V_{z0} = 11$ operand Z not used (Reduction CVA)

E_T: Enable bit for early termination
C_T: Condition code for early termination

p_op: Opcode for primary arithmetic
s_op: Opcode for secondary arithmetic

(b) PVA Instruction



Vector_Opcode: Opcode for vector instructions

$E_0/E_1/E_S$: Enable bits for data streams L_0, L_1 and S, respectively

E_T: Enable bit for early termination
C_T: Condition code for early termination

cs-store-index: Index of the instruction in the loop body that will initiate a cs-store operation

Loop_size: Size of loop body in number of instructions

Figure 4.6: Format of CVA and PVA Instructions

back to R3. The CVA instruction, in this example, will have the following settings: V_{x1}/V_{x0}=01, V_{y1}/V_{Y0}=10, $V_{z1}=V_{z0}$=11, E_S=0, E_T=0, C_T="don't care".

In this example, if the vector executions is interrupted, register R3 will contain the partial result of the inner product computations. Upon returning from the interrupt, computations will continue from where it was left off. □

Example 4.8:

Implement the vector arithmetic: C[i] = sA[i], for all i.

```
Initialize CR to the vector length.
Initialize SSR[STR0] to stride value for vector A.
Initialize SSR[STRS] to stride value for vector C.
Initialize R0 to the starting address for vector A.
Initialize R2 to the starting address for vector C.
Initialize R5 to s.
CVA     mul @L0, R5, @S;
```

This is a compound CVA. The second arithmetic is a "no-op". In this case, L_0 and S are enabled, but not L_1. Prior to the vector executions, R5 was initialized with the constant s. The CVA instruction, in this example, will have the following settings: V_{x1}/V_{x0}=01, V_{y1}/V_{y0}=00, V_{z0}/V_{z1}=00, E_S=1, E_T=0, C_T="don't care". □

4.5 Terminating Conditions

All vector executions (CVA and PVA) can exit their executions and return to a scalar mode prior to CR becoming zero. A vector instruction (a CVA or a PVA instruction) can optionally enable this capability by setting the E_T bit to one, and by specifying this early terminating condition in the C_T field (see Figure 4.6 on page 61). During the course of vector executions, if the condition code equals to those specified in the C_T bit, the vector executions will terminate *immediately.*

4.5.1 Early Termination for CVA Executions

The p_op field, in this case, specifies some scalar arithmetic that will alter the condition code. This is the only early terminating condition for CVA executions.

4.5.2 Early Termination for PVA Executions

In PVA executions, conditional and unconditional branches are allowed within a loop body. When a branch inside the loop body is taken, and the target of the branch lies outside the loop body, then the execution of the PVA loop is considered to terminate.

A PVA loop can terminate its executions via one of the following three mechanisms.

(1) The CR reaches 0; or

(2) When a branch resides within the loop body is taken and the target of the branch lies outside the loop body; or

(3) The early termination capability is explicitly enabled (by setting the E_T bit in the PVA instruction) and during the course of the PVA executions, the condition code is set to those specified in the C_T field.

Conditions (2) and (3) above are collectively referred to as the early termination for PVA executions. That is, they terminate prior to CR reaches zero.

When executing the last instruction of the loop body, if the last instruction does not cause a change-of-control flow with a target lies outside the loop body, then the loop execution will continue and the control is transferred back to the top of the loop. Example 3.5 on page 48 illustrates examples for using the early exit mechanisms (2) and (3) described above.

4.6 Register Overlay

In this Section, we will introduce the notions of *Register Overlay* and *Temporary Register.* These notions are applicable to PVA executions only.

In vector arithmetic, most data loaded from the memory via the constant-stride loads are *temporaries* only, in the sense that they are consumed in a single iteration, and are never used again. Registers R6 and R7 shown in Example 1.1 on page 9 are examples of such temporaries.

Furthermore, if a vector arithmetic is allowed to be interrupted, then all the constant-stride load/store addresses associated with the vector executions need to be saved so that the load/store operations can resume upon returning from the interrupt. Storing all the prefetched temporaries from the memory as well as these load/store addresses using some architectural visible storage spaces (such as the general purpose register file or control registers) could be an inefficient use of

these valuable resources.

A new approach, called *Register Overlay*, is introduced to address this problem. In this approach, upon entering a PVA execution mode, a portion of the architectural visible register file is "overlaid" with a new set of registers. When a register is being *overlaid*, it has two instances: (i) an *overlaid instance*; and (i) a *temporary instance*. When in the PVA mode, only its temporary instance is visible to a programmer, but not its overlaid instance. Conversely, when the execution exits the PVA mode and enters a scalar mode, the overlaid instance becomes visible again and the temporary instance cease to exist.

Figure 4.7 shows how the visibility of the overlaid registers change over a sequence of three execution modes: scalar, PVA and scalar.

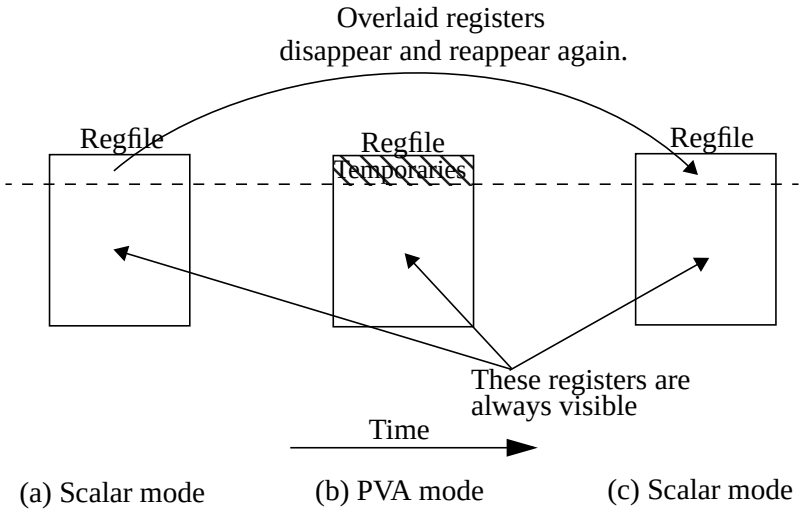


Figure 4.7: Register Overlay

In this work, registers R0 and R1 are designated as the set of registers that could be overlaid during PVA executions. They are shown in Table 4.2. The overlaid instances of these registers are used to store the corresponding cs-load addresses. The temporary instances of these registers are used to store the data prefetched from the memory via the cs-load L_0 and L_1.

Table 4.2: Overlaid and Temporary Instances of R0/R1

Registers	Overlaid Instance (Only Accessible In Scalar Mode)		Temporary Instance (Only Accessible In PVA Mode)	
	Contents	Accessibility	Contents	Accessibility
R0	Load address for L_0	Read/Write	Prefetched data for L_0	Read Only
R1	Load address for L_1	Read/Write	Prefetched data for L_1	Read Only

The temporary instances of R0 and R1, respectively associated with L_0 and L_1, are read-only registers. Writing to these temporary registers within the loop body are ignored by the hardware. These temporary registers are only defined during the PVA executions. When a PVA loop exits its executions and enters a scalar mode, the data contained in these temporary registers are lost. Access to such a register, at that point, will retrieve the overlaid instance of the register, which is the load address for the last cs-load operation performed.

Also, when a PVA execution is interrupted, these temporary registers are not saved as part of the context. Upon returning from the interrupt, the cs-load operations that prefetched the data into the temporary registers will be re-initiated, using the load addresses stored in the overlaid instances of R0 and R1. The temporary instances of R0 and R1 will be re-initialized before the normal PVA executions can resume.

4.7 Machine States Maintenance For Vector Executions

In this machine, all vector executions are interruptible, in the sense that an interrupt could cause a temporary suspension of a vector execution, even before the entire vector arithmetic is completed.* Besides the usual context for scalar executions, certain additional contexts will need to be saved so that the vector executions can properly resume later on.

When an interrupt occurs, some of these vector contexts are saved and some are discarded. In this Section, we will discuss the minimum vector contexts that need to be saved to maintain machine states consistency with fast interrupt response time.

4.7.1 Saving The Execution Modes

A vector execution mode is indicated by having a non-zero value in the Count Register (CR). Conversely, having a zero value in the CR indicates that the machine is (or had been) executing in a scalar mode.

In addition, the content of IXR is used to distinguish between the CVA and PVA execution modes: for CVA executions, IXR is always zero; for PVA executions, it is always non-zero.

Upon returning from an interrupt, the IXR and CR are both examined by the hardware, in order to put the machine in the appropriate execution mode prior to resuming the normal execution.

---

* In this context, interrupts and exceptions are equivalent concepts.

4.7.2 Saving The Minimum Vector Contexts

Upon an interrupt, the vector contexts that will be saved by the hardware are shown in Table 4.1. They include SSR, CIR, the overlaid instances of R0 and R1, and register R2 through R6. Since these registers are already in the Regfile, they are automatically saved along with all other registers in the Regfile. A shadow register file similar to those used in [SHARC97, TMS320C3x, MCORE98] can be used to achieve fast interrupt response time.

CVA Executions

For CVA executions, operations on each vector element is *atomic,* in the sense that if the result of the operations associated with a vector element is not written back when an interrupt occurs, then *all* the intermediate results will be discarded. All operations performed on this element (or elements) will have to be repeated upon returning from the interrupt.

When performing a reduction or hybrid CVA, the partial result is constantly written back into the Regfile (to R3). If the CVA is interrupted, the partial result is already in the Regfile and is automatically saved. No additional time is wasted to save the partial result. When returning from the interrupt, however, the content of R3 will need to be restored back onto the s_dbus before normal CVA executions can resume.

PVA Executions

For PVA executions, all the intermediate results produced in the loop body are stored in the Regfile. The machine, in this case, maintains its consistency at the boundaries of the scalar instructions within a loop body. Thus no additional time is wasted to save the intermediate results.

The temporary instances of R0, R1, for PVA executions, are not saved as part of the vector contexts. Upon returning from the interrupt, the cs-loads (L_0 and L_1) that fetched these temporaries, if enabled, are re-initiated. Temporary registers R0 and R1 are then updated accordingly before the normal PVA executions can resume. The hardware, in this case, assumes that the memory locations have not been altered during the course of servicing the interrupt.

When a PVA instruction is decoded, a copy of the PC is saved in a temporary hardware location. When an interrupt occurs during the PVA executions, the saved copy of the PC (pointing to the PVA instruction) is restored and saved as part of the vector context. When returning from the interrupt, using this PC, the PVA instruction is fetched to recover all the loop control information, including cs-store-index, Loop_size, etc. The content of IXR is then added to the PC to obtain the address of the instruction in the loop body where the execution is to be resumed.

In PVA executions, the execution of the instruction located at the “cs-store” label, and its associated cs-store operation is an atomic operation. Consider the PVA loop shown in Example 4.6 on page 56. The “movt” instruction and the associated cs-store operation is an atomic operation. If a cs-store does not complete due to an exception or an interrupt, then the “movt” instruction is also considered “not executed”. Upon returning from the interrupt, executions will resume starting at the “movt” instruction.

4.7.3 Updates of Temporary and Overlaid Instances of R0 and R1

When a PVA instruction is decoded and begins its executions, the first two cs-load operations are initiated immediately (if they are enabled). The first read from R0 (or R1) in the loop will yield the first data loaded via L_0 (or L_1); the second read from R0 in the loop will yield the second data element loaded via L_0 (or L_1), and so on. It is possible to have multiple reads from R0 (or R1) within a single iterations.

During PVA executions, at any given point in time, the temporary instance of R0 (or R1) always contains the data fetched from memory via L_0 (or L_1) using the load address stored in the overlaid instance of R0 (or R1). In terms of machine states consistency, the updates of the temporary and overlaid instances of R0 (or R1) occur simultaneously in a lock-step manner: they are both updated at the instruction boundary between the instruction that reads R0 (or R1) and the subsequent instruction in the program order.

4.8 Memory Organization

A simplistic view of the memory organization of this machine is shown in Figure 4.8. In this machine, there are three independent on-chip memory modules: M0, M1, and TM (Temporary Memory). In addition, there is a small instruction cache, called the *loop cache*, for storing program loop instructions during PVA executions.

M0 and M1 are the main on-chip memories. M0 is used to store instructions and data. M1 is used to store data only. TM is also used to store data only. In particular, it is used to store temporary vectors during vector executions.

In this memory system, the load unit L_0 has read access to M0 and TM; the load unit L_1 has read access to M1 and TM; the store unit S has write access to all M0, M1 and TM. M0 and M1 are single ported memories. TM has one read port and one write port. The contents, accessibilities and the number of read and write ports of these memory modules are shown in Table 4.3.

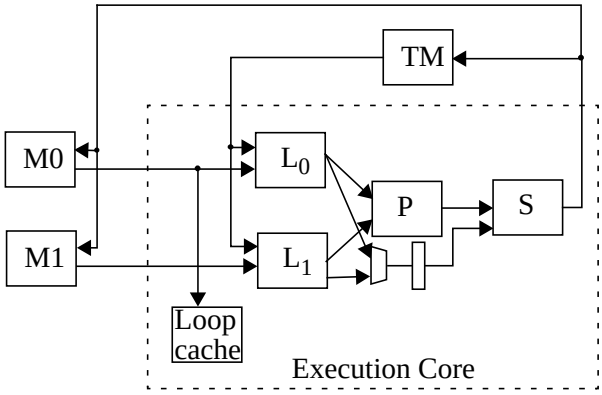


Figure 4.8: A Simplistic View Of The Memory Organization

Table 4.3: Accessibilities of M0, M1 and TM

Memory Module	Contents	Data Streams			Number of Read/ Write Ports	Arbitrate Between Streams
		L0	L1	S		
M0	Instructions and data	Read	-	Write	1 (read or write)	L_0 vs. S
M1	Data	-	Read	Write	1 (read or write)	L_1 vs. S
TM	Data (temporary vectors)	Read	Read	Write	2 (one read and one write)	L_0 vs. L_1

4.8.1 Memory Bandwidth Requirements For Vector Executions

To perform all the CVA arithmetic shown in Figure 1.5 on page 9, with a sustain throughput rate of one, we require the memory system to support two data reads and one data write. There is no instruction request during a CVA execution. This is because once a CVA instruction is decoded and executed, no further instruction is needed for the rest of CVA executions.

In addition to the two data reads and one data write, the PVA executions also require one instruction to be fetched in each cycle.

In each cycle, the memory system shown in Figure 4.8 can support up to two data reads and one data write (through M0, M1 and TM); it can also support one instruction fetch in the same cycle (using the Loop Cache).

4.8.2 Memory Map For M0, M1, TM

The three memory modules M0, M1 and TM can be accessed by referencing certain pre-defined memory space (i.e. they are memory mapped modules). The memory map for M0, M1 and TM is shown in Figure 4.9.

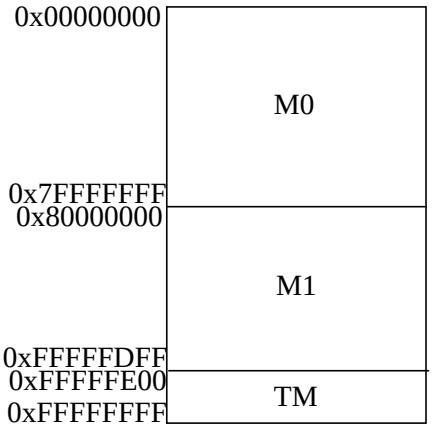


Figure 4.9: Memory Map For M0, M1, TM

4.8.3 Temporary Memory

TM is a small RAM memory used for storing temporary vectors during vector executions. It can also be used to store some frequently used constant vectors (such as the coefficient vectors in digital filtering).

TM is an extension of vector registers in the traditional vector machines for holding temporary vectors. Like those on traditional vector machines, the optimizing compilers attempt to operate on these temporary vectors as much as possible prior to writing them back to the memory. TM helps reduce the memory bandwidth pressure on M0 and M1. It also helps reduce the power consumptions on these larger memory modules.

There are two major differences between TM and vector registers.

- Accesses to TM are made by referencing the appropriate memory space, instead of explicitly specified in the vector instructions (as vector register numbers). In particular, these accesses are made by setting up the data streams L_0, L_1 and S.
- When constructing, allocating and utilizing these temporary vectors, the compilers have more flexibilities in organizing the temporary storage space. For example, if a TM can store a vector of *n* elements, then it can also be organized as a storage space for *m* vectors, each with a length of *n/m* elements. The TM can also be organized as a storage space for multiple vectors with different length. The compilers, in this case, can manage the vector allocations to minimize fragmentations within TM.

For the purpose of this work, TM is assumed to be 512 bytes, direct-mapped, with one read port and one write port. The following example illustrates how TM can be utilized to speedup the vector executions.

Example 4.9:

Illustrate how the executions of the vectorized loop shown in Example 4.4 on page 53 can benefit from using TM.

Recall that this loop is performing: C[i] = (lsr(A[i], r9) | lsl(A[i], r8). The vectorized loop is shown below.

```
<Some initialization code>
// assign L0 to A; assign S to a temporary vector in TM.
mov r5, r9
CVA     lsr @L0, r5, @S;
<Some initialization code>
//assign L0 to A; assign L1 to the temporary vector in TM
//assign S to vector C.
mov r5, r8
CVA     lsl @L0, r5, @P,     or @L1, @P, @S;
```

In this example, a temporary vector is created and allocated in TM. The destination of the first CVA instruction and one of the source operands of second CVA instruction access the temporary vector through TM. The first CVA instruction sources vector A from M0 via L_0 and writes the temporary vector to TM via S. The second CVA instruction sources vector A again from M0 via L_0 and sources the temporary vector from TM via L_1. It also writes the result vector to M1 via S. The execution activities for these two CVA instructions are shown in Figure 4.10.

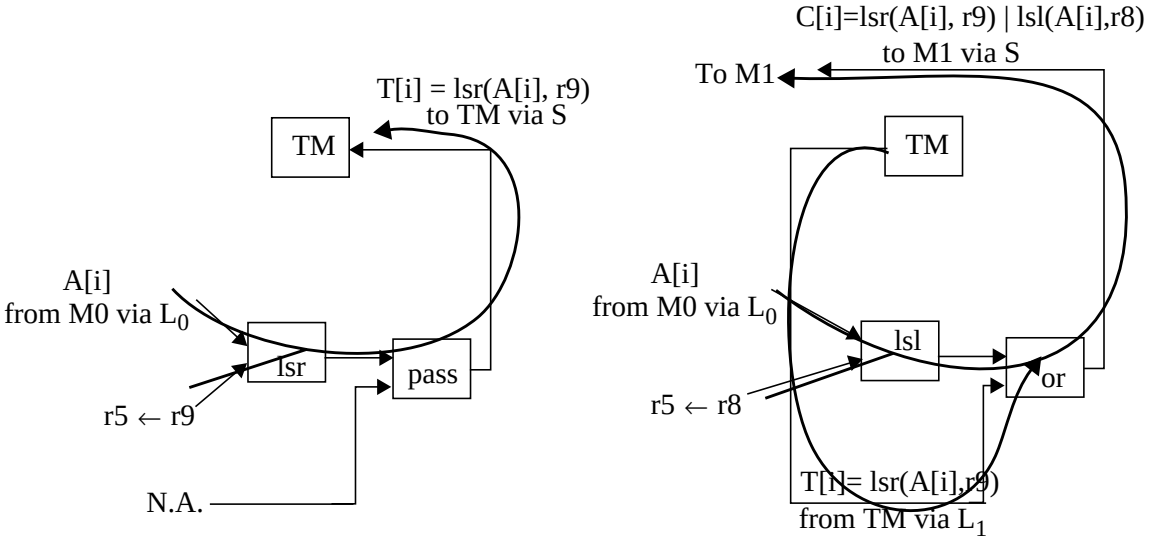
In this example, the second CVA instruction uses three data streams (two inputs and one output). No memory conflict arises in these executions. Using M0 and M1 alone would have caused a lot of memory conflicts. □

4.8.4 Strip Mining For TM

When the size of a vector being processed is larger than the size of TM, the vector operations are broken down, under software control, into multiple vector operations, with each of them operates on vectors with length that can fit into TM. This is the TM equivalence of *strip-mining* for vector registers [Cray1,Patterson96].

Unlike the fixed length vector registers, however, the compilers, in this case, have the flexibilities to trade-off between the number of temporary vectors it can allocate and utilize, and the number of strip-mined iterations. This concept will be illustrated later in Example 4.11.

Example 4.10:



(a) Execution activities for the first CVA instruction

(b) Execution activities for the second CVA instruction

Figure 4.10: Execution Activities For The Two CVA Instructions

Strip-mine the vectorized code shown in Example 4.9, assuming that the vector length is not known at compile time.

In this example, there are only two CVA instructions and *one* temporary vector involved, it is possible to have the entire TM dedicated to storing a *single* temporary vector. Since each vector element is four bytes long (a word) and the TM is 512 bytes, a vector with length greater than 128 elements will require some strip-mining code to "wrap around" the vector instruction to avoid overflowing the TM. The following shows the strip-mined code, in C-style language, for a source vector with an unknown length, n [Patterson96].

```
low = 1;
VL = (n mod 128);            // find the odd size piece first
for (j=0; j<n/128; j++) {
   for (i=low; i<low+VL-1; i++) {  // runs for length VL
      C[i] = (lsr(A[i], r9)   |   lsl(A[i], r8)); //main op.
   }
   low = low + VL;
   VL = 128;                // reset VL to 128 after the first
                            // odd size piece
}
```

□

The following example illustrates how TM could be used to reduce power consumption, while maintaining the highest possible performance level.

Example 4.11:

Vectorize the following reduction operations: $\sum_{i = 0, n-1}$((A[i]*B[i]+C[i]) * A[i]*B[i]*D[i]), for some independent vectors A, B, C and D. Assume that vectors A and C reside in M0; vectors B and D reside in M1.

An optimum solution, in terms of execution time, using 3 temporary vectors and 4 CVA instructions (3 compound CVA and 1 reduction CVA), is shown below.

(1) T1[i] = A[i] * B[i];
(2) T2[i] = T1[i] + C[i];
(3) T3[i] = T1[i] * D[i];
(4) Reduction result = $\sum_{i = 0, n-1}$ (T2[i] * T3[i]);

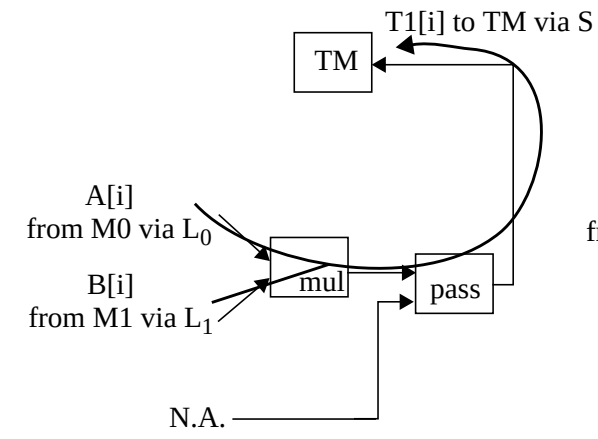
Since vectors A and B reside in M0 and M1, T1 must be allocated in TM. Thus vector T3 must be in M0 (given that D resides in M1). Since vector C resides in M0, T2 can be in M1 or TM. Table 4.4 shows two possible solutions for allocating the temporary vectors.

Table 4.4: Two Possible Solutions For Allocating The Temporary Vectors T1, T2 and T3

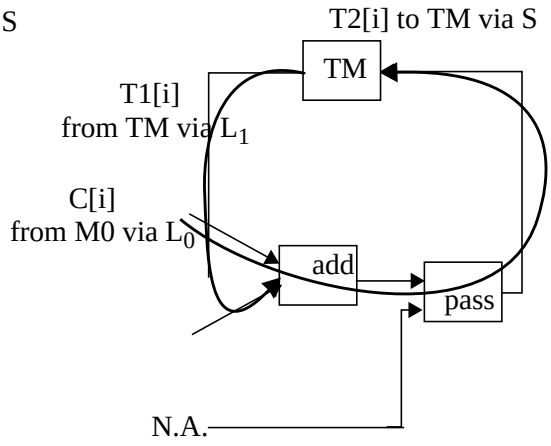
Temporary Vectors	Solutions (I)			Solution (II)		
	M0	M1	TM	M0	M1	TM
T1			X			X
T2		X				X
T3	X			X		

Both solutions have no memory conflict and thus have the same performance level. Solution (II), however, provides a lower power solution since T2 is allocated in TM instead of in M1.

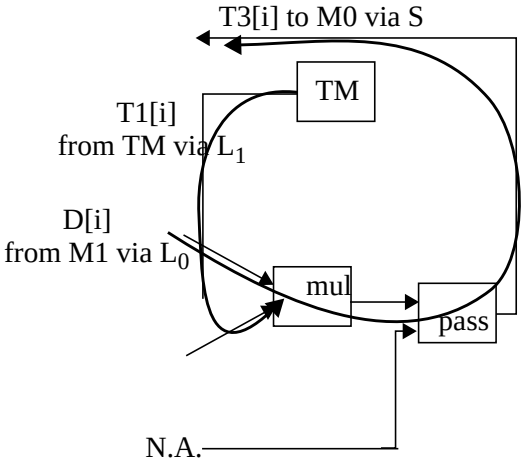
A drawback of Solution (II) is that it requires the two temporary vectors T1 and T2 to reside in TM simultaneously. If TM is not big enough to hold both vectors, then Solution (I) is the only viable solution. If the TM is too small for even a single vector, then Solution (I) will need to be strip-mined. Figure 4.11 shows the execution activities for the four CVA instructions, using Solution (II). □



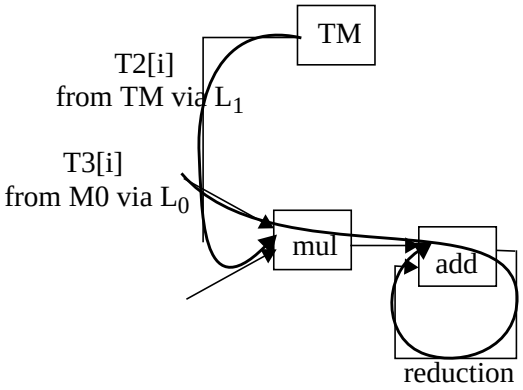
(a) Execution activities for the first CVA instruction



(b) Execution activities for the second CVA instruction



(c) Execution activities for the third CVA instruction



(d) Execution activities for the fourth CVA instruction

Figure 4.11: Execution Activities For The Four CVA Instructions Using Solu tion (II)

CHAPTER 5

PSEUDO-VECTOR MACHINE IMPLEMENTATIONS

In this Chapter, we will discuss how we could implement the pseudo-vector machine.

5.1 Datapath Implementations

The CPU architecture of this machine is shown in Figure 5.1.

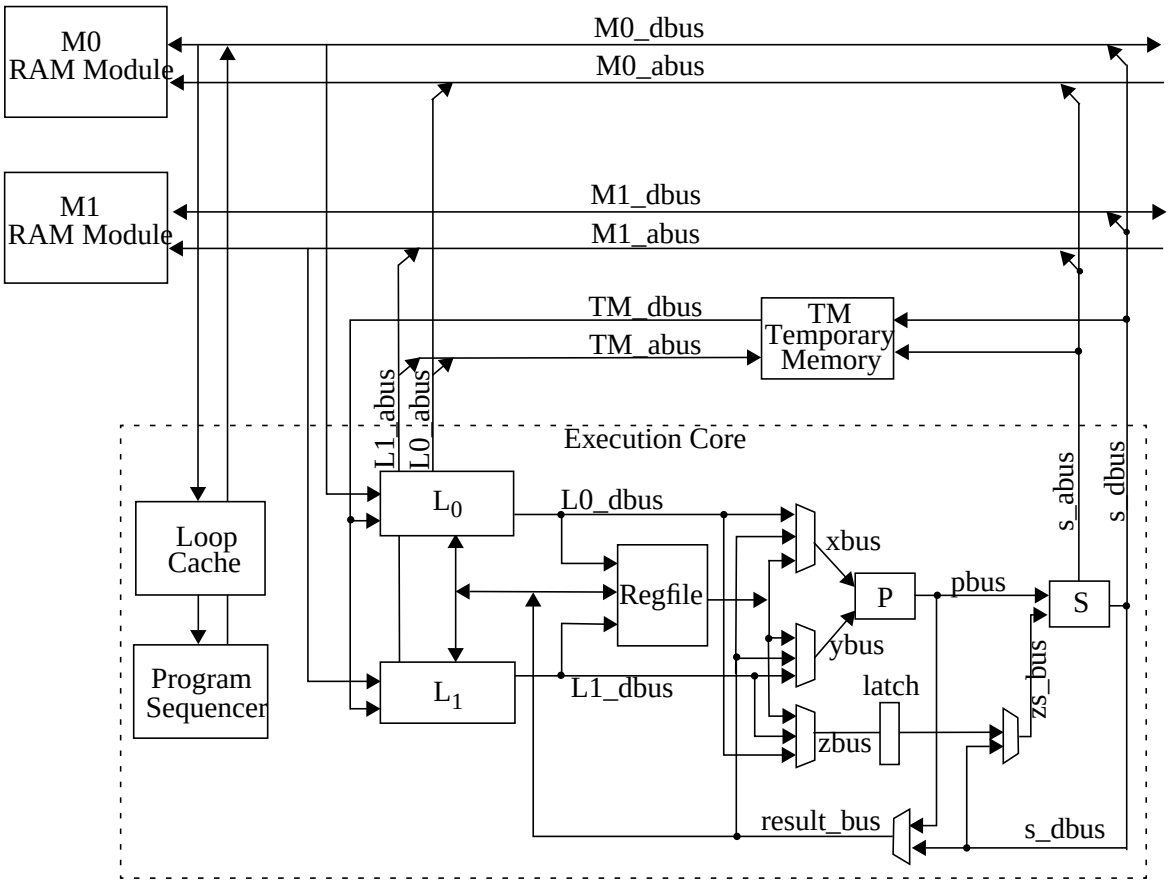


Figure 5.1: The CPU Architecture for the Pseudo-Vector Machine

We will begin by comparing our datapath implementation with a single-issued, four-stage pipelined machine. The datapath the latter machine is shown in Figure 5.2. In this datapath, the P unit denotes a general purpose functional unit that is capable of performing all the arithmetic functions defined in the ISA. These functions include add, shift, logical, multiply, load, store, etc.

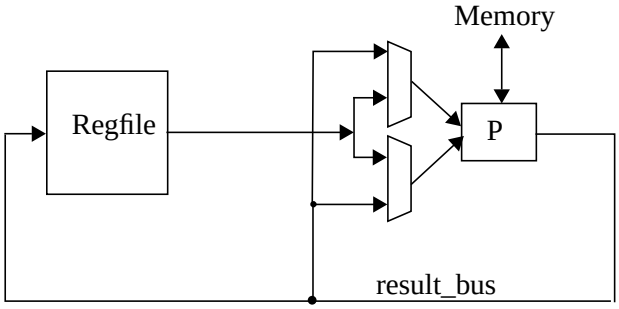


Figure 5.2: Datapath for a Single-Issued, Four-Stage Pipelined Machine

The datapath for our proposed pseudo-vector machine is shown in Figure 5.3. The P unit in this datapath is similar to those in the single-issued machine, except that the memory load and store functions have been moved to the L_0, L_1 and S units. Furthermore, all multi-cycle arithmetic functions in the P unit are fully pipelined, including the integer multiply.* For the purpose of this work, an integer multiply is assumed to take two cycle to execute.

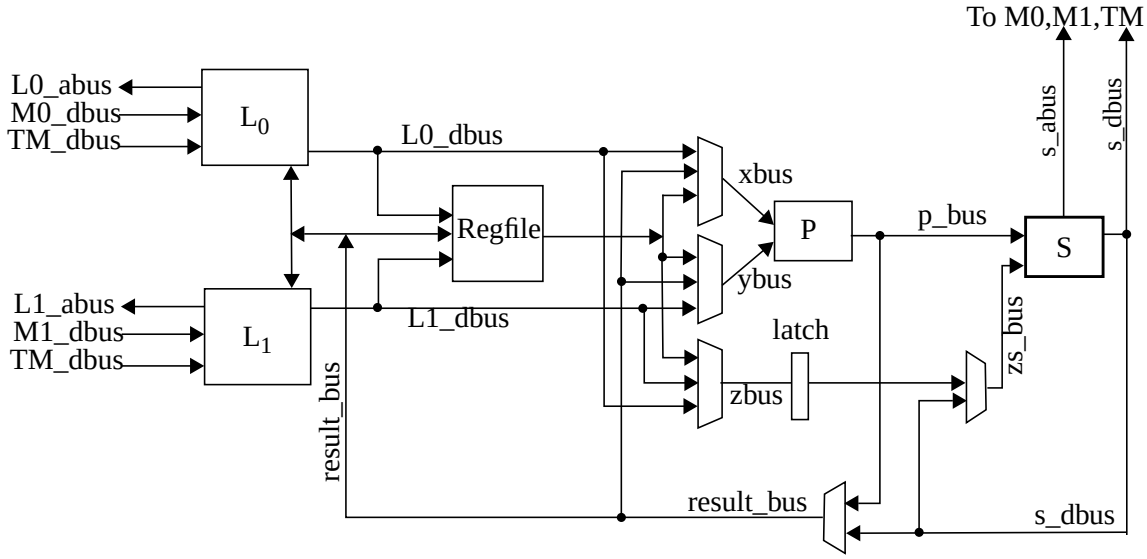


Figure 5.3: Datapath For The Pseudo-Vector Machine

The result_bus in this datapath is now driven by either the P unit or the S unit. This bus is used for writing back results to the Regfile, or feed-forward to the P unit. Comparing these two fig-

---

* In this work, we will only focus on integer arithmetic. Floating point arithmetic is beyond the scope of this dissertation.

ures, the following load/store units have been added to the pseudo-vector machine:

- **L_0 and L_1 Units**. These are memory load units. These units fetch data from the memory (M0, M1 or TM) and load them into the Regfile and/or feed-forward to the P unit. Within the L_0 (L_1) unit, there is a local copy of register R0 (R1). The L_0 (L_1) unit constantly snoops the result_bus for any write back activity to register R0 (R1) and updates its local copy of the register.

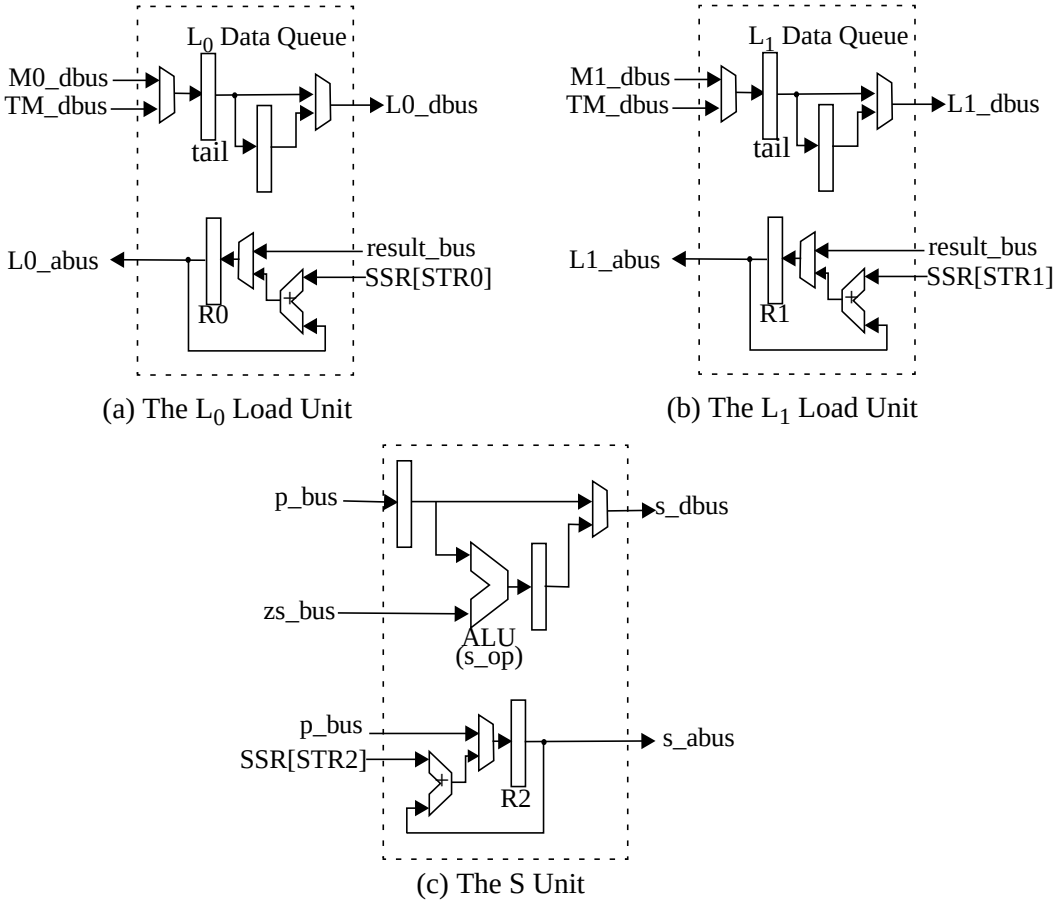
- **S Unit**. This is a memory store unit. This unit performs store operations to the memory (M0, M1 and TM). Besides the store operations, the S unit can also perform the s_op arithmetic. These arithmetic include some simple, commutative arithmetic and logical functions such as “add”, “and”, “or”, “xor”, etc. Within the S unit, there is a local copy of register R2. The S unit constantly snoops the p_bus for any write back activity to register R2 and updates its local copy of the register.

Notice that the four functional units (namely, the L_0, L_1, P and S units) are chained together to form a Y-shape datapath. In addition, a feedback path (s_dbus and zs_bus) exists between the output and the input of the S unit. This feedback path is used for accumulating the partial results for reduction and hybrid CVA. The basic structure of this datapath is to implement *all* the CVA shown in Figure 4.2 on page 51.

The implementations of these load/store units are depicted in Figure 5.4. Within the L_0 (or L_1) unit, there is a two-deep data queue and a local copy of register R0 (or R1). The data queues in L_0 and L_1 also have an extra by-pass path that allows the data in the tail of the queue to feed directly into the Regfile. Likewise, the S unit contains a local copy of register R2.

The local copies of registers R0, R1 and R2 within the L_0, L_1 and S units can be automatically updated by the hardware by adding by the stride values SSR[STR0], SSR[STR1] and SSR{STR2], respectively. These updates occur if the corresponding data streams are enabled during the CVA or PVA executions. At any given point in time, the local register R0 (or R1) of L_0 (or L_1) unit contains the latest load address, and its corresponding data is stored (or is to be stored) at the tail of the queue (see Figure 5.4).

The S unit, on the other hand, has a data queue of only one-deep. When a load request (from L_0 or L_1 unit) has a memory conflict with a store request (from the S unit), the latter will take precedence.



(a) The L_0 Load Unit

(b) The L_1 Load Unit

(c) The S Unit

Figure 5.4: The Implementations of L_0, L_1 and S Units

In this datapath, both scalar and vector execution modes use the *same* P unit. This means that many arithmetic functions that are available to the scalar executions are also available to the vector executions.

When performing cs-load and cs-store operations, the hardware does not disambiguate memory references. i.e. it does not check for any data hazard associated with the memory references. It is the responsibility of the vectorizing compilers or assemblers to ensure that all the elements in the source and destination vectors are independent in the memory.

5.2 Scalar Executions

When executing in a scalar mode, the machine operates like a single-issued machine. It reads operands from the Regfile, executes in the P unit, and writes back to the Regfile and/or feed -forwards to the P unit for the next computation. It can also load data from the memory, via the L_0 unit and the L0_dbus, to the Regfile and the P unit. It can also store data to the memory using the S unit.

5.3 CVA Executions

When executing in a CVA mode, data are continuously streamed from the memory via the L_0 and/or L_1 units, into the P unit; the results are then optionally written back to the memory via the S unit (for compound and hybrid CVA).

To perform a compound CVA, consider a vector operation described by C[i] = sA[i]. The L_0 unit shown in Figure 5.3 loads the source vector A from the memory, an element at a time, and feeds it into the P unit. At the same time, the constant s is read from the Regfile. The P unit then performs the multiply operations, and the S unit performs the cs-store operations.

To perform a reduction CVA, consider a vector inner product described by $\sum_{i = 0, n-1}$(A[i]*B[i]) (see Example 4.7 on page 60). Instead of performing the cs-store operations, the S unit performs the “add” function and accumulates the partial results via the s_dbus and zs_bus. At the same time, these partial results are constantly written back to R3 in the Regfile, via the result_bus.

To perform hybrid CVA, in addition to accumulating the partial results, the S unit also constantly writes the results to the memory via a cs-store operation. In this case, the S unit writes to two destinations in each cycle: R3 in the register file and the memory via S.

For reduction and hybrid CVA, the partial result is initialized as follows. When a result is *first* produced on the p_bus (by the P unit), the S unit performs a “pass” function on this bus and drive the data directly onto the s_dbus, instead of performing the s_op function (see Figure 5.4(c)). This result is used as the initial value for the partial result. This operation corresponds to the initialization of the partial sum, S_0, described in Section 3.1.2 and Section 3.1.3 on page 41.

5.4 PVA Executions

When executing in a PVA mode, the datapath behaves as if it is executing in a scalar mode, except that: (i) if the L_0 (L_1) stream is enabled, the L_0 (L_1) unit prefetches data from the memory and loads them into the temporary instance of R0 (R1) in the Regfile; and (ii) if the S stream is enabled, the S unit stores data to the memory, using the data produced by the P unit. The PVA executions will be illustrated using the following example.

Example 5.1:

Vectorize the program loop shown in Example 3.1 on page 44.

This is a program loop with multiple exit points. In this example, two cs-loads are performed during each iteration. The loop is controlled by a loop index i, for i=0,...,n-1. In addition, if B[k]>A[j], for some j and k, the loop will exit early. This loop can be vectorized using our PVA loop construct as follows.

```
Initialize R3 to m.
Initialize CR to n.
Initialize SSR[STR0] to 4; SSR[STR1] to 8.
Initialize R0 to the starting address for vector A;
Initialize R1 to the starting address for vector B;

PVA    @L0,@L1,#loop_size;
complt R3, R0                  // Is (A[j] > m)?
bf     EXE_N                   // Branch if false
[Instructions for the “M” block]
br    CONT                     // unconditional branch
EXE_N:
[Instructions for the “N” block]
CONT:
complt R0, R1                  // Is (A[j]<B[k])?
bt EXIT                        // Exit if true. Otherwise,
                               // branch back to top of loop
EXIT:
```

In the vectorized loop, both the L_0 and L_1 streams are enabled while S is not. In the loop body, accesses to A[j] and B[k] are made by reading from the temporary registers R0 and R1, respectively.

Upon entering the loop executions, two data are fetched from the memory via the L_0 and L_1 units and loaded into the tails of their respective data queues. In the subsequent cycle, two additional data are fetched from the memory while the data already in the data queues are moved into the temporary registers R0 and R1 in the Regfile, respectively.

During subsequent iterations, each time when R0 (or R1) is read in the program loop, a data is moved from the data queue in the L_0 (or L_1) unit into the temporary register R0 (or R1), while

another cs-load is initiated by the L_0 (or L_1) unit to fill the tail of its data queue.

The overlaid instance of R0 (or R1) in the Regfile is constantly updated by the hardware with the corresponding load address. At any given point in time, the temporary instance of R0 (or R1) in the Regfile contains the data prefetched from the memory, with an address that is stored in the overlaid instance of R0 (or R1) in the Regfile. The same correspondence can be said about the contents of the tail of the data queue in the L_0 (or L_1) unit and its local copy of R0 (or R1).

Figure 5.5 shows an example of how the local copy of R0 in L_0 unit and the overlaid instance of R0 in Regfile are being updated when streaming in a vector. It shows that the former (contains the address of the data currently being fetch) always runs ahead of the latter (contains the address of the data being committed into the Regfile).

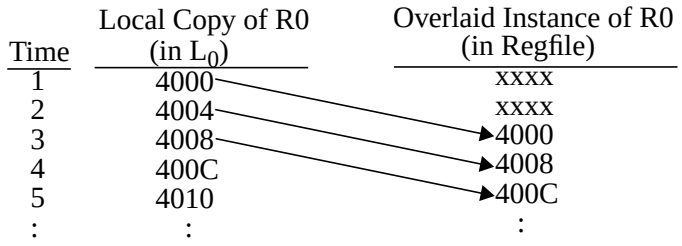


Figure 5.5: Updates of Local R0 in L0 and Overlaid Instance of R0 in Regfile

Notice that there are multiple conditional and unconditional branches within the PVA loop body. In this example, there are two possible ways to exit this loop: one is by CR reaches zero; the other is when the last instruction “bt EXIT” is taken. If this last branch instruction is not taken, the control will be transferred back to the top of the loop and the PVA executions will continue. □

5.5 Managing The PVA Loop Executions

To monitor the loop execution and the target of a branch, a counter based scheme similar to those proposed in [Lee99a,Lee99b,Lee99d] can be used.

When a PVA instruction is encountered, the Loop_size specified in the instruction is captured by the hardware. In addition, the IXR register is used to keep track of which instruction within the loop body is currently being executed. This register behaves like a “local PC” within the loop body. When the first instruction in the loop is being executed, IXR is set to one. For each instruction sequentially executed, this register is incremented by one. When the last instruction in the loop is being executed, IXR is set to the Loop_size.

When the IXR equals to Loop_size-1, the instruction fetch will be directed towards the first instruction of the loop. If the last instruction is a sequential instruction or the instruction does not cause a change-of-control flow with a target lies outside the loop body, the execution will transfer back to the beginning of the loop and IXR is reset to one.

When a branch is taken during a PVA execution (either in a forward or a backward direction), the branch displacement field of the branch instruction is added to the IXR register. When the taken branch is in a forward direction, the branch displacement field is a positive number. When this value is added to IXR, the latter will point to the target somewhere further down the branch instruction. When the taken branch is in a backward direction, the branch displacement field is a negative number. When this value is added to IXR, the latter will point to the target somewhere before the branch instruction.

With these branch adjustments, the IXR will always point to the correct instruction relative to the PVA instruction. After a taken branch, if the resulting IXR is non-negative or if the IXR is greater than the Loop_size (indicating that the branch target is located outside the loop body), the PVA loop execution will terminate.

Figure 5.6 shows an implementation of the IXR register. The IXR is incremented by one for each instruction executed sequentially in the loop body. When a branch is taken, the displacement field of the branch instruction is added to IXR.

The IXR_count is constantly compared with zero, Loop_size and Loop_size-1. If (IXR_count≤ 0) or (IXR_count>Loop_size), the PVA executions will terminate immediately. If (IXR_count == Loop_size-1), an instruction fetch from the top of the loop will be initiated.

This counter based approach is very similar to the Enhanced Scheme proposed in [Lee99b,Lee99d]. Interested readers are referred to that reference.

If a PVA execution terminates with CR equals zero, no cycle penalty is incurred. Towards the end of the last iteration of a PVA execution, if the last instruction in the loop is a taken conditional branch with its target lies outside the loop body, a cycle penalty will be incurred for wrongfully fetching and executing the first instruction in the loop. In this case, the result of executing this instruction will be squashed (i.e. the result will be discarded and not written back). A new instruction fetch using the branch target will be initiated.

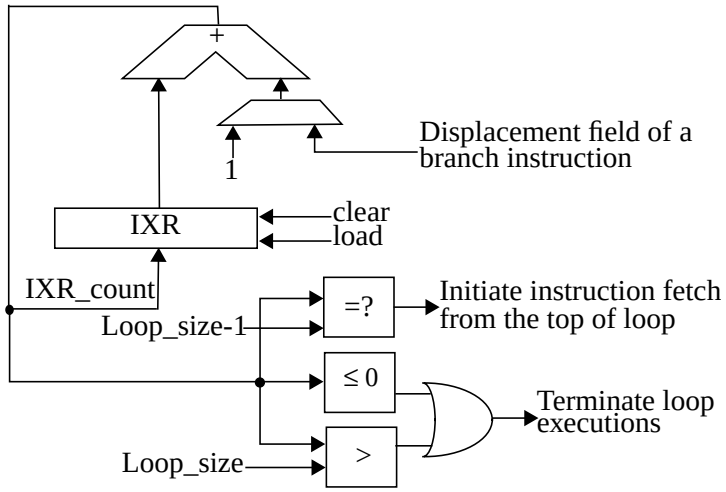


Figure 5.6: The IXR Register

5.6 Implementing The Temporary Registers

Figure 5.7 shows an implementation of the Regfile. This Regfile contains, besides other registers, an overlaid and temporary instances of R0 and R1 (O_R0, T_R0, O_R1 and T_R1 respectively).

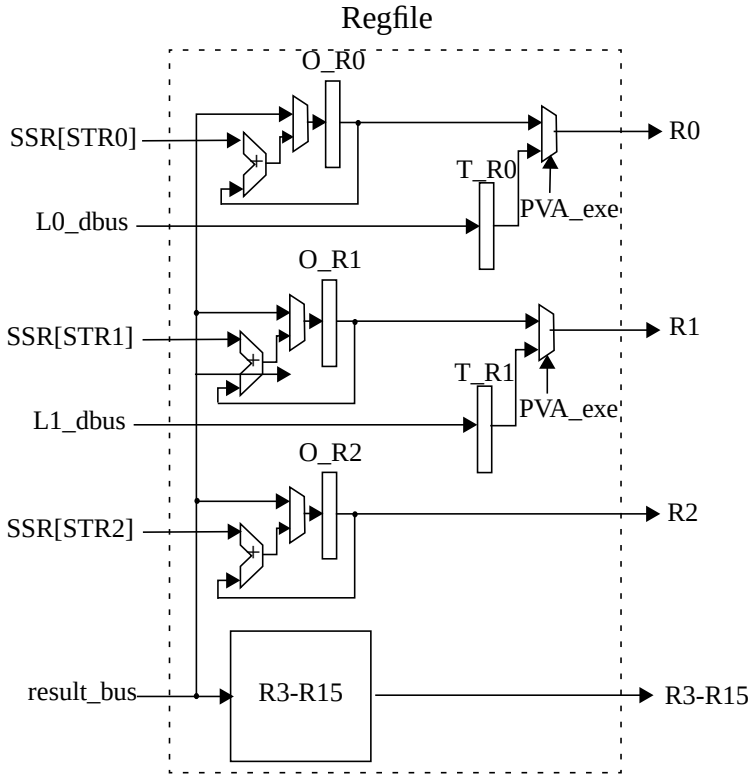


Figure 5.7: Register File with Temporary and Overlaid Instances of R0 and R1

O_R0, O_R1 and R2 can be updated by adding the stride values SSR[STR0], SSR[STR1]

and SSR{STR2], respectively. These updates occur when the cs-load or cs-store operation is committed (to the Regfile or to the memory). At any given point in time, T_R0 (or T_R1) contains the data prefetched from the memory, with an address stored in O_R0 (or O_R1).

This figure also shows how reading from registers R0 and R1 are selected between their temporary instance and overlaid instance, depending on whether we are in PVA mode.

5.7 Implementing The Memory System

The memory system of this machine is very similar to those implemented in the SHARC ADSP-21061 chip [SHARC97], except that this machine has an additional independent memory module: the Temporary Memory (TM).

The memory organization of this machine is shown in Figure 5.1 on page 74. In this machine, there are three independent on-chip memory modules: M0, M1, and TM. In addition, there is also a loop cache for storing program loop instructions during PVA executions. All these memory modules are typically SRAM memories.

These three memory modules can be accessed by the L_0, L_1 and S streams by referencing certain pre-defined memory address space (i.e. they are memory mapped modules). The mapping for M0, M1 and TM is shown in Figure 4.9 on page 69.

5.8 Loop Cache For Storing PVA Program Loops

For CVA executions, instruction fetch is not necessary since once the machine enters a CVA execution, it no longer requires any instruction.

For PVA executions, a small loop cache is used to store the program loops. This loop cache will first attempt to eliminate any access conflicts at M0 with data references. Once this is achieved, it will also attempt to capture the entire program loop to reduce access power at M0.

For the purpose of this work, we will assume that the loop cache is organized as a 32-entry (64 instructions), direct-mapped cache. Each loop cache entry stores two 16-bit M-CORE instructions. Each instruction request, on this machine, fetches two 16-bit M-CORE instructions.

All scalar instructions in a PVA loop body can be classified into two categories. When an instruction fetch causes a conflict with data references at M0, the two instructions being fetched are called *Essential Instructions*. Otherwise, they are called *Non-Essential Instructions*.

Due to the pipeline nature of the machine, a load instruction that accesses M0 (or an instruction that sources R0 in PVA executions), I_i, can only cause a conflict at M0 with two instruction fetches down the program order. That is, data fetch for I_i can only conflict with a fetch request for instructions: (i) I_{i+3} and I_{i+4}, if I_i is at odd address; or with (ii) I_{i+4} and I_{i+5}, if I_i is at even address.* The following illustrates the case where I_i is odd address aligned.

```
I_i     // This instruction reads R0 and is odd address aligned.
I_i+1
I_i+2
I_i+3  // cause conflict at M0 - an essential instruction
I_i+4  // cause conflict at M0 - an essential instruction
```

Due to the "wrap-around" nature of the loop, I_i can also be at the bottom of the loop and I_{i+3}, I_{i+4} and I_{i+5} at the top of the loop.

Upon entering a PVA execution, when allocating instructions into the loop cache, priority is first given to essential instructions. After all these instructions are allocated, the loop cache will then try to allocate as many non-essential instructions as possible, into the loop cache, in an attempt to reduce access power at M0.

The loop cache operates as follows. In addition to a valid bit, there is an E bit (*Essential bit*) associated with each entry in the loop cache. During the first four iterations of the loop, both the loop cache and M0 are accessed in parallel for all instruction requests.

During the first two iterations of the loop, when an instruction request causes a conflict at M0 *and* the requested instructions are not found in the loop cache, an entry is allocated in the next cycle for the two essential instructions associated with this request. Their corresponding E bit is set.

During the third iteration of the loop, the loop cache will then try to capture as many non-essential instructions as possible. It does so by replacing the existing non-essential entries (entries with their E bits cleared).

During the fourth iteration, the following two conditions (or events) are monitored closely by the hardware:

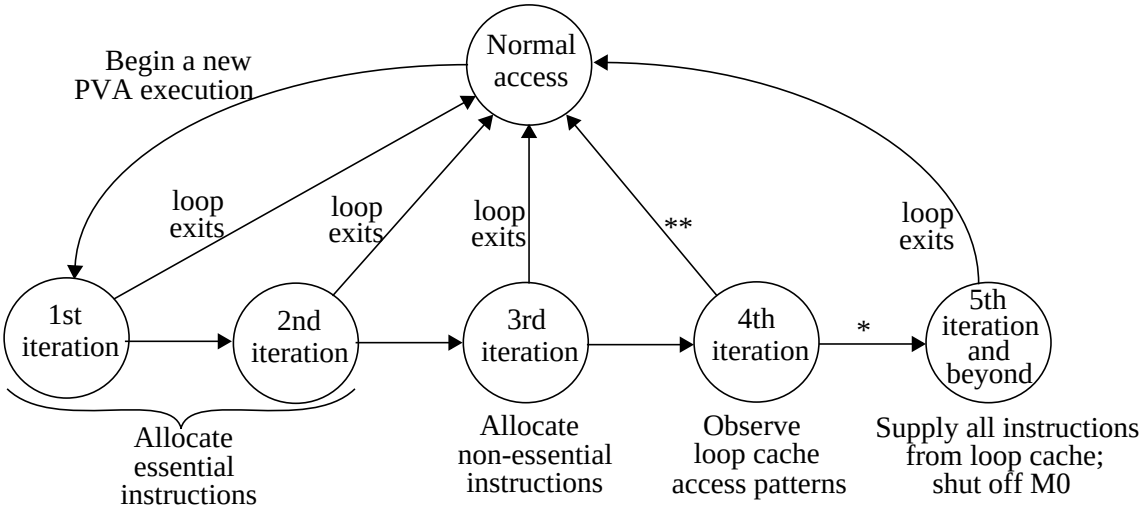
---

* Assuming a big endian address mode.

- All instruction requests are found in the loop cache;
- There is no change-of-control flow instruction inside the loop body.

If both of these conditions are satisfied during the fourth iteration, then from the fifth iteration and beyond, all instructions are supplied by the loop cache, avoiding *any* instruction access to M0.

The state machine for the loop cache controller is shown in Figure 5.8. It shows that during the first two iterations, the loop cache is warmed up with all the essential instructions. During the third iteration, it is warmed up with the non-essential instructions, as much as possible. If the loop cache is successful in capturing the entire loop, as determined during the fourth iteration, from the fifth iteration and beyond, M0 can be shut off completely for all instruction requests.



* (all instruction accesses during the 4th iteration hit in the loop cache) &&
(no change-of-control flow instruction found in the loop body) &&
no loop exits

** Not of condition "*"

Figure 5.8: Loop Cache Controller

Note that it is possible to have essential instructions from different program loops to co-exist in the loop cache - an essential entry can only be replaced by another essential entry. In this work, a loop cache flush occurs whenever there is a context switch.

If the loop cache is filled with many essential instructions, then there is little room in the cache for non-essential instructions. In this case, very little M0's instruction access power can be saved. Under any circumstances, a non-essential instruction will *not* replace an essential instruc-

tion already in the loop cache. That is, performance consideration always supersedes the power consideration.

Interested readers are referred to some other related work on instruction caching for small program loops [Lee99a,Lee99c].

CHAPTER 6

BENCHMARK CHARACTERISTICS AND PERFORMANCE EVALUATION METHODOLOGIES

In this Chapter, we will describe the benchmark programs we use in this work. We will describe some critical loop related characteristic. We will then describe our experimental methodologies for evaluating the performance benefits for using the pseudo-vector machine. The actual performance results will be given in Chapter 7. All performance results, in this dissertation, will be given relative to a single-issued, four stage pipeline scalar machine, which we call the *base machine*.

We will first define two metrics for performance evaluations.

6.1 Metrics For Performance Evaluations

We define *performance improvement* as

$$\text{Performance Improvement} = \frac{E_S - E_V}{E_S} \tag{6.1}$$

where E_S is the execution cycles for the original scalar program, and E_V is the execution cycles for the vectorized program. We define *speedup* as

$$\text{Speedup} = \frac{E_S}{E_V} = \frac{1}{1 - \text{Performance Improvement}} \tag{6.2}$$

In the rest of this dissertation, we will use these two metrics to quantify the performance benefits for using the pseudo-vector machine.

6.2 Benchmark Programs And Their Characteristics

The benchmark programs used in this work, called the PowerStone benchmarks, is shown in Table 6.1. These benchmarks were compiled to the M•CORE ISA using the Diab 4.2.2 compiler. The number of dynamic instructions executed and the number of instructions being fetched on the base machine are also presented in this Table.

Table 6.1: PowerStone Benchmarks

Benchmark	Dynamic Inst. Executed	Dynamic Inst. Fetched	Descriptions
auto	17374	20695	Automobile control application
blit	72416	78448	Graphics application
compress	322101	355216	A Unix utility
des	510814	519037	Data Encryption Standard
engine	955012	1058154	Engine control application
fir_int	629166	705966	Integer FIR filter
g3fax	1412648	1681130	Group three fax decode
g721	231706	256025	Adaptive differential PCM for voice compression
jpeg	1342076	1528812	JPEG 24-bit image decompression standard
map3d	1228596	1463233	3D interpolating function for automobile control applications
pocsag	131159	147202	POCSAG communication protocol for paging applications
servo	41132	42919	Hard disc drive servo control
summin	1330505	1532825	Handwriting recognition
ucbqsort	674165	804662	U.C.B. Quick Sort
v42bis	1488430	1660493	Modem encoding/decoding

Percentage Execution Time On Critical Loops

In this work, performance improvement is achieved by vectorizing all the critical loops found in a benchmark program. Thus, it is important to closely examine what percentage of execution time the program spends on executing these loops. By Amdahl's law, *the performance will be limited by the fraction of time the program spends in non-loop executions.*

Table 6.2 below shows, for each benchmark, the number of critical loops and the percentage of execution time it spends on these loops. Due to time constraints and the large number of loops that exist in these benchmark programs, as a rule of thumb, any program loop that consumes less than 2-3% of the overall execution time is not typically considered as *critical*.

Table 6.2: Percentage Execution Time Spent In Program Loops

Benchmarks	Number of Critical Loops	Number of Iterations per Invocation	% Execution Time In Program Loops (%)
auto	2	702.5	71.76
blit	2	1000	99.18
compress	1	7969	9.81
des	2	9.31	2.61
engine	2	3.51	9.77
fir_int	1	33.6	69.74
g3fax	2	960.4	56.65
g721	1	5	2.87
jpeg	5	33.39	49.02
map3d	1	13.5	37.93
pocsag	2	18.3	61.21
servo	0	N.A.	0.00
summin	5	173.9	81.52
ucbqsort	1	2.27	0.58
v42bis	1	8271	2.48
Average	1.73	1371	39.65

Figure 6.1 shows the percentage of execution time a program spent on critical loops, in graphical form. Depending on the benchmarks, this percentage varies greatly from 0% for `servo` to 99% for `blit`. Five out of the fifteen benchmarks spent less than 5% of their execution times in loops.

Seven out of the fifteen benchmarks spent less than 10% on loops. The other eight benchmarks spent, on average, 66% of their times in loops. On average, a benchmark spends about 37.9% on a few handful of critical program loops - a significant but not an overwhelming fraction of execution time.

If these benchmark programs spend *zero* execution time on these critical loops (i.e. with an *infinite* speedup during loop executions), then the average performance improvement will be about *37.9% - this represents the upper limit on our overall performance improvement number.*

We will present our performance enhancement results in two ways: (i) performance improvements over the entire original scalar program executions; and (ii) performance improvements during loop executions (i.e. the time the programs spent in non-loop executions is not considered).

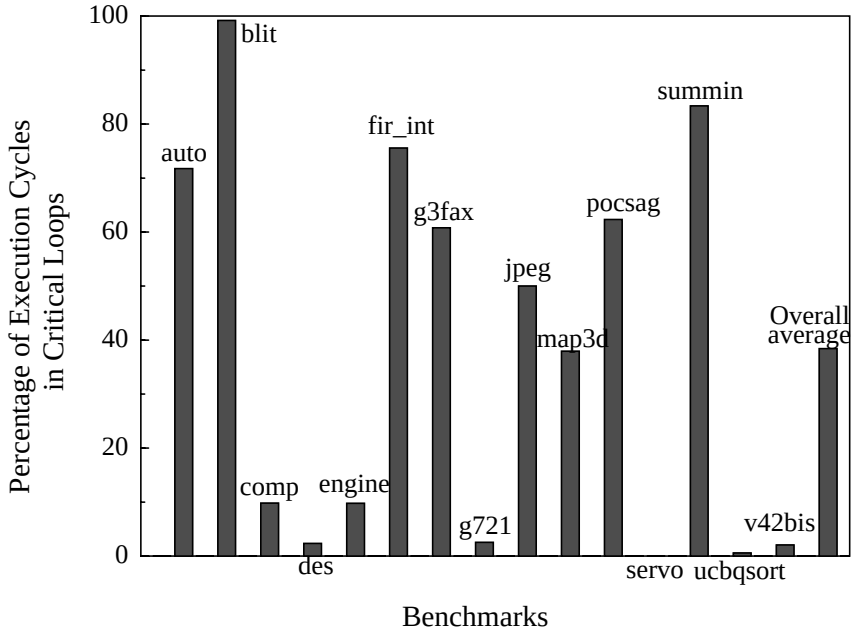


Figure 6.1: Percentage of Execution Cycles Spent in Critical Loops

Table 6.2 also shows the average number of iterations per loop invocation. The higher the average number of iterations per invocation, the less impact the vector startup costs will have on the overall performance. From the above Table, this number also varies greatly across the benchmarks, from a few iterations to thousands of iterations per invocation.

6.3 Performance Evaluation Methodologies - Overview

Developing a vectorizing compiler for this pseudo-vector machine is beyond the scope of this work. Without a vectorizing compiler, there is no vectorized assembly code. Without which, it is impossible to evaluate exactly the performance benefits by using a detail simulation model of the machine. Instead, the following approach is adopted.

The benchmarks were not re-compiled to vectorize the critical loops. Cycle-based simulations were first performed on a single-issued, four-stage pipelined machine (or the base machine). This base machine does not have any vector processing capability. The performance statistics collected on this scalar machine were used as a base result. All performance improvement numbers for various vector processing techniques will be given relative to this base result.

The original scalar programs were dynamically profiled. Each program loop in these benchmarks was marked; the number of invocations and the number of iterations were recorded.

We then vectorize these critical loops by hand, at the assembly level, either by using the

CVA instruction(s) and/or the PVA instruction(s). The number of cycles saved for each loop are then computed using the profiled statistic. Vector setup and exit costs (described in the next Section) are subtracted from these savings. We then sum up the net savings for each loop to give the total saving.

Depending on the types of experiments, we may vectorize the loops: (i) strictly using the PVA constructs only; or (ii) strictly using the CVA constructs only; or (iii) Using both constructs. For the case of (iii), we will choose a solution by using between CVA and PVA constructs, or a combination of both, whichever that will yield the best performance result.

In doing so, we obtain three set of experimental results, which we will call, respectively,

- *CVA-only executions*;
- *PVA-only executions*; and
- *CVA/PVA executions*.

The results given by “CVA-only executions” loosely track the performance achievable by a conventional 2-deep vector machine. The results given by “PVA-only executions” loosely track the performance achievable by a *single-issued* DSP machines*. The results given by “CVA/PVA executions” represent the performance achievable by a machine that is capable of executing in both “true” vector mode and “pseudo” vector mode.

6.4 Vector Setup and Exit Costs

In this Section, we will breakdown all the overheads associated with the vector executions. We will estimate and assign a fixed cost to each of these overhead components. The setup and exit costs for vector executions can be broken down as follows.

- Special registers initialization costs;
- Vector instruction decoding costs;
- Pipeline warm-up costs;
- Vector mode exit costs;
- Initial access conflicts at M0 (for PVA executions only).

6.4.1 Special Registers Initialization Costs

Depending on the type of vector executions, some or all of the registers listed in Table 6.3

---

* The performance of a conventional DSP VLIW machine is not evaluated in this work.

need to be appropriately initialized prior to vector executions. The additional setup costs to initialize these registers are listed in this Table. These additional costs are *in addition* to the setup costs already incurred in the scalar version of the program loops, prior to vectorization.

Table 6.3: Additional Registers Initialization Costs

Registers	Register Contents	Additional Setup Costs (cycles)[a]
SSR	Stride values and operand sizes for L_0, L_1 and S	3
CIR	Number of iterations to be executed	1
R0	Load address for L_0	1
R1	Load address for L_1	1
R2	Store address for S	1
R3,R4,R5	Scalar constants for CVA executions	1

a. If applicable

Take R0/R1/R2, for example. These registers need to be initialized to the starting addresses for various data streams, if they are enabled. In the scalar program loops, certain general purpose registers will also need to be initialized to these addresses, although they are not necessarily R0/R1/R2. The vectorizing compiler, in this case, can rename the last writes of these registers to R0/R1/R2 appropriately; if successful, the *additional* vector setup costs for initializing these registers are zero. In this work, we will assume that an additional “mov” instruction is *always* needed to initialize each of these registers.

For CVA executions, both SSR and CIR are always initialized. For PVA executions, depending on the loop, only one of SSR and CIR, or both, need to be initialized.

SSR is a special control register and can be initialized using the following M-CORE instruction sequence. In this work, all stride values for the cs-load and cs-store operations are assumed to be known at compile time.

```
lrw r3,[Stride_Sixe_Vector] //load Stride_Size_Vector into R3
mtcr r3, SSR                // move r3 to SSR
       :
Stride_Size_Vector:
.long     0xXXXXXXXX       // The actual stride/size vector
```

The two-instruction sequence is assumed to take three cycles to execute: two cycles for the

first instruction and one cycle for the second.

CIR is also a special control register. In the scalar version of the program loops, the iteration count (or the vector length), if used, need to be initialized to a general purpose register. For our pseudo-vector machine, the additional setup costs for initializing CIR is the *extra* "mtcr" instruction. This instruction is assumed to take one cycle to execute.

R3 through R5 are used for compound CVA. If used, they need to be initialized to the required scalar constants. To initialized each of these registers, we assume that an extra "mov" instruction is needed.

6.4.2 Vector Instruction Decode Costs

We assume that two cycles are needed to: (i) decode the vector instruction; and (ii) initiate the first cs-load operations from M0 and/or M1.

6.4.3 Additional Pipeline Warm-Up Costs

For CVA executions, due to the chaining of the P and S units, additional pipeline warm-up costs are incurred before the first result of the CVA computations is available. If the p_op performed at the P unit takes one cycle to execute, and the s_op performed at the S unit takes zero cycle to execute, then the *additional* pipeline warm-up cost is zero.

In general, the additional pipeline warm-up costs for CVA executions is given by t_p+t_s-1, where t_p is the execution time for p_op; and t_s is the execution time for s_op. They are both are expressed in number of cycles. t_p ranges from one cycle (all except the multiply function) to two cycles (the multiply function); while t_s ranges from zero cycle (by-passing the S unit) to one cycle (an ALU arithmetic).

These costs are only associated with CVA executions. For PVA executions, this cost is zero.

6.4.4 Vector Mode Exit Costs

Upon exiting a vector mode, the PC needs to be adjusted and the instruction fetch needs to be appropriately redirected. We will assume that the machine incurs one cycle penalty for exiting a vector mode.

6.4.5 Initial Access Conflicts At M0

This cost is associated with PVA executions only. Instruction and data references can cause conflicts at M0 during loop executions. When such a conflict arises, a stall cycle is incurred and the instructions that cause this conflict are captured into the loop cache in the subsequent cycle.

The allocations of these essential instructions into the loop cache occur only during the first two iterations of the loop. Once all the essential instructions are captured, they will remain in the loop cache until they are replaced by other essential instructions (see Section 5.8 on page 83).

If I_i has a data reference that accesses M0, then it will only cause access conflict with the instruction request for I_{i+3} and I_{i+4}, or with I_{i+4} and I_{i+5}, depending on the address alignment of I_i. Furthermore, due to the wrap-around nature of the loop, I_i can be near the bottom of the loop and I_{i+3} and I_{i+4} (or I_{i+4} and I_{i+5}) can be near the top of the loop. The loop cache typically uses the first two iterations to capture all the essential instructions in the loop.

6.5 PVA-Only Executions - Three Types Of Loop Execution Overheads

In a PVA-only execution, the number of execution cycles saved can be categorized into the following categories. The overall cycle saving will be given by the sum of each of these categories.

- Cycle saving due to eliminating the loop control overheads (lp-ctl-oh);
- Cycle saving due to eliminating the cs-load overheads (cs-load-oh);
- Cycle saving due to eliminating the cs-store overheads (cs-store-oh).

We will denote these loop execution overheads as *lp-ctl-oh*, *cs-load-oh* and *cs-store-oh*, respectively. The lp-ctl-oh refers to the time spent on executing the branch instruction at the end of the loop, as well as the loop index increment/decrement instruction. The cs-load-oh and cs-store-oh refer to the time spent on executing the load/store instructions, as well as their associated load/store address increment/decrement instructions. The use of a PVA construct is precisely aiming at eliminating these three types of loop execution overheads.

There are vector setup and exit costs associated with PVA executions as described Section 6.4. For the purpose of calculating these overheads, the setup and exit costs for a given PVA execution are divided *equally* among all of the categories of overheads involved, whenever applicable. This is illustrated in the following example.

Example 6.1:

Vectorize the loop shown in Example 3.3 on page 46 using a PVA loop. Also calculate the saving due to eliminating the various types of loop overheads. Assuming that this loop is executed 100 times and the branch instruction “bt EXIT” was never taken.

For convenience sake, this loop is illustrated below.

```
L1:
ld.h        r7,(r2)   // load A[i]
addi        r2,2
ld.h        r6,(r3)   // load B[i]
addi        r3,2
cmplt       r6,r7     // is A[i] > B[i]?
bt          EXIT      // if so, exit the loop
decne       r14       // if not, decrement the count
bt          L1        //is the entire vector being processed?
                      //if not, branch backward
EXIT:
```

The CVA version of this loop is shown in Example 3.4 on page 48. This loop can also be vectorized using a PVA loop as follows.

```
Initialize CR appropriately.
Initialize both SSR[STR0] and SSR[STR1] to 2.
Initialize R0 and R1 to the appropriate starting addresses.
L1:
PVA     @L0, @L1, #2; // A PVA loop with 2 scalar instructions
cmplt   R1, R0
bt      EXIT
EXIT:
```

This PVA loop only enables L_0 and L_1. The original loop took 11 cycles per iteration. The “bt L1” is taken 99 times and not taken 1 time. During the last iteration, the “bt L1” takes only one cycle to execute. The total execution time of the original loop is: 11 x 99 + 10 x 1 = 1099.

The setup and exit costs for the PVA loop is: 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit) = 9 cycles. The PVA loop body, which consists of two scalar instructions, takes 2 cycles to execute. Thus the total cycles for PVA executions is: 9 + 2 x 100 = 209 cycles. Compared to the scalar execution, a saving of 1099 - 209 = 890 cycles. The speedup is given by 1099/ 209=5.26.

By enabling L_0 and L_1, we can eliminate the first four instructions in the original loop. These four instructions take 6 cycle per iteration to execute. They are associated with the cs-load-oh. Thus the saving due to eliminating the cs-load-oh is 6 cycles per iteration, or a total of 600 cycles.

The PVA loop also successfully eliminated the last two instructions in the original loop. These two instructions take 3 cycle per iteration to execute. They are associated with the lp-ctl-oh. Thus the saving due to eliminating the lp-ctl-oh is 3 cycles per iteration, except for the last iteration, where the saving is only 2 cycles. Thus the total saving for eliminating lp-ctl-oh is 99 x 3 + 1 x 2 = 299.

There is no instruction in this loop that is associated with the cs-store-oh. Since the setup and exit cost is 9 cycles. This cost will be equally divided between cs-load-oh and lp-ctl-oh, with 4.5 cycles each. Thus the final saving due to eliminating the cs-load-oh is 600 - 4.5 = 595.5 cycles; the final saving due to eliminating the lp-ctl-oh is 299 - 4.5 = 294.5 cycles. The total saving is given by 595.5(for cs-load-oh) + 294.5(for lp-ctl-oh) = 890 cycles.

The performance improvement due to eliminating various types of overheads are shown in Table 6.4. The overall speedup for PVA-only executions, in this case, is 1099/209=5.26. □

Table 6.4: Performance Improvements Due to Eliminating Various Types of Loop Execution Overheads

Types of Overhead Eliminated	Cycle Saving	Performance Improvements	Overall Speedup
lp-ctl-oh	294.5	0.2680	-
cs-load-oh	595.5	0.5418	-
cs-store-oh	0	0.0000	-
Total	890	0.8098	5.26

Example 6.2:

Vectorize the program loop shown in Example 4.1 on page 51 using the PVA construct only. Estimate the execution cycles saving due to eliminating the various types of loop overheads when the loop is executed 100 times.

This loop can be vectorized using the PVA loop construct as follows.

```
Initialize R5 to the appropriate constant.
Initialize CR appropriately.
Initialize SSR[STR_S] to 4.
Initialize R2 to the appropriate starting address.
L1
PVA     @S, #1;         // PVA loop with 1 scalar instruction
mov R2, R5              // initiate cs-store S
```

The original program loop took 6 cycles per iteration (or 6 x 99 + 5 x 1 = 599 cycles total); while the vectorized loop took only 1 cycle per iteration (not including the setup and exit costs).

The vector setup and exit costs are: 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R3) + 1(exit) = 8 cycles. This cost will be divided equally between lp-ctl-oh and cs-store-oh.

In this loop, only the S stream is enabled. By enabling S, we could eliminate the “stw” and “addi r14,4” instructions, a saving of 3 cycles per iteration. In order to initiate a cs-store operation in each iteration, however, a “mov R2, R6” instruction is added to the loop body. Thus for cs-store, the saving is actually 2 cycles per iteration. Thus the total saving for eliminating cs-store-oh is 2 x 100 - 4 = 196 cycles.

Likewise, the PVA loop construct can eliminate the branch overheads by eliminating the “decne” and “bt” instructions, also a saving of 3 cycles per iteration, except for the last iteration where the saving is 2 cycles. Thus the total saving for eliminating lp-ctl-oh is 3 x 99 + 2 x 1 - 4 = 295.

Total saving: 196(for cs-store-oh) + 295(for lp-ctl-oh) = 491 cycles; or a speedup of 599/(599-491)=5.55.

Alternatively, the PVA loop took 1(one instruction) x 100(iterations) + 8(setup/exit costs) = 108 cycles to execute. Total saving = 599 - 108 = 491 cycles. □

Table 6.5: Performance Improvements Due to Eliminating Various Types of Loop Execution Overheads

Types of Overhead Eliminated	Cycle Saving	Performance Improvements	Overall Speedup
lp-ctl-oh	295	0.4925	-
cs-load-oh	0	0.0000	-
cs-store-oh	196	0.3272	-
Total	491	0.8197	5.55

6.6 Cycle Saving Calculations for Vectorizing a Typical Scalar Loop

In this Section, we will derive a generic expression for calculating cycle saving for a vectorizing *typical* scalar loop. Consider the following loop.

```
L1:
        I0
```

```
I1
:
br   L1
```

A “typical loop” here refers to a loop: (i) that exits only through the branch at the end of the loop (i.e. through “`br` L1” not taken); and (ii) there is a loop index increment/decrement instruction within the loop body. A “typical loop” always exits with the register CR becoming zero. That is, the loop does not exit early.

For loops that are not “typical”, the cycle saving calculations are more ad hoc and will not be described here. We will first define the following terms.

n_1 be the number of times “br” is taken;
n_2 be the number of times “br” is not taken;
n be the total iteration count for the scalar loop ($n=n_1+n_2$);
p be the number execution cycles for the primary arithmetic performed at the P unit;
O_l be the cs-load-oh per iteration (cs-load-oh = $(n_1 + n_2) * O_l$);
O_s be the cs-store-oh per iteration (cs-store-oh = $(n_1 + n_2) * O_s$);
O_{lp} be the lp-ctl-oh per iteration, when “br” is taken (lp-ctl-oh = $(n-1) * O_{lp} + 1 * (O_{lp}-1)$);
C_{PVA} be the vector setup and exit cost for the PVA executions;
C_{CVA} be the vector setup and exit cost for the CVA executions;
t_{sc} be the execution cycles per iterations for the scalar loop, when “br” is taken;
t_p be the number execution cycles for the primary arithmetic performed at the P unit;
t_s be the number execution cycles for the secondary arithmetic performed at the S unit;

For CVA executions, the initial vector setup and exit costs will be $C_{CVA}=C_{PVA}+t_p+t_s-1$. That is, for $t_p=1$ and $t_s=0$, the setup and exit costs for both CVA and PVA executions are identical.

6.6.1 Saving Calculations For Typical PVA Executions

For a “typical loop” described above, O_{lp} =3 cycles: 2 cycles for eliminating the execution of the taken branch instruction and 1 cycle for eliminating the execution of the loop index increment/decrement instruction. But when the branch is not taken, the saving is only 2 cycles. Thus the cycle saving due to removing lp-ctl-oh is typically given by: $3 * n_1 + 2 * n_2$. *

For a typical PVA execution, saving due to removing cs-load-oh = $O_l*(n_1+n_2)$; saving due to removing cs-store-oh =$O_s*(n_1+n_2)$; saving due to removing lp-ctl-oh =$3*n_1 + 2*n_2$.; the total

---

* This rule, however, does not hold if the loop index is also used somewhere else in the loop body. In this case, the loop index increment/decrement instruction can not be removed from the PVA loop body.

setup/exit cost = $C_{PVA}*n_2$. Thus the total saving is given by:

$$\text{Total PVA saving} = (O_l + O_s) * (n_1 + n_2) + (3*n_1 + 2*n_2) - C_{PVA} * n_2 \quad (6.3)$$

6.6.2 Saving Calculations For Typical CVA Executions

When a "typical" loop is vectorized using a CVA construct, the resulting CVA executions always terminate with the count register CR becoming zero. Furthermore, since the original scalar loop is invoked n_2 times, the CVA instruction is also executed n_2 times.

When a loop is CVA vectorizable, the per-iteration execution time of the loop, t_{sc}, can be divided into five components: O_l, O_s, O_{lp}, t_p, t_s, where O_{lp} is again 3 cycles for the "typical" loop. Thus, t_{sc}, for the scalar loop, in this case, can be written as

$$t_{sc} = O_l + O_s + O_{lp} + t_p + t_s = O_l + O_s + 3 + t_p + t_s$$

The total execution cycles for the scalar loop, E_s, is given by

$$E_s = t_{sc} * n_1 + (t_{sc}-1) * n_2 = (O_l + O_s + 3 + t_p + t_s) * n_1 + (O_l + O_s + 3 + t_p + t_s - 1) * n_2$$

A CVA execution incurs an initial setup cost and an actual vector execution cost. For n_2 executions of a CVA instruction, the initial setup cost amounts to $C_{CVA}*n_2$. Since the original loop is executed $n=n_1+n_2$ iterations, n results are being produced in the equivalent CVA executions. Assuming there is no memory conflict during the CVA executions and the sustained throughput rate is one, the actual vector execution cost is equal to $n=n_1+n_2$.

Thus for n_2 executions of the CVA instruction, the total execution time is given by

$$E_v = (C_{CVA} * n_2) + (n_1+n_2) = (C_{PVA}+t_p+t_s-1) * n_2 + (n_1+n_2)$$

Thus, for CVA execution,

CVA saving = $E_s - E_v$; or

$$\text{CVA saving} = (O_l+O_s) * (n_1+n_2) + (t_p+t_s+2)*n_1 + 2*n_2 - C_{PVA}*n_2 \quad (6.4)$$

Notice that equations (6.3) and (6.4) are identical when $t_p=1$ and $t_s=0$. Comparing these two equations, CVA executions will outperform PVA executions if t_p is greater than 1 and $t_s=0$,. This is particularly true when n_1 is relatively large and n_2 is relatively small.

Example 6.3:

Estimate the cycle saving for the loop shown in Table 6.6 for: (i) using PVA construct only; (ii) using CVA construct only.

This loop, taken from "auto" benchmark (loop number 1), can be vectorized by using the PVA-only construct, and by using the CVA-only construct, as shown in Table 6.6.

Table 6.6: "auto" Critical Loop 1

Assembly Instructions	Using PVA Construct Only	Using CVA Construct Only
0000010d2 stw r5,(r14) 0000010d4 addi r14,4 0000010d6 decne r6 0000010d8 bt 0x010d2	PVA @S, #1 cs-store: mov r5, r5	CVA mov r5, @S;

Table 6.7 shows the profiled statistics for the loop. They include the number of time "bt" is taken (n_1=704) and the number of time "bt" is not taken (n_2=1).

Table 6.7: Profile For Critical Loop 1

Address	Entry Type	Exe. Counts, $n=n_1+n_2$	Branch Target	Taken count, n_1 (%)	Not taken count, n_2 (%)
0000010d2 0000010d8	target bt	705 705	- 000010d2	- 704 (99.9)	- 1 (0.142)

For PVA executions, O_l=0, O_s=2, O_{lp} = 3. O_s is 2 cycles instead of 3 because to perform the cs-store, an additional "mov" instruction is introduced into the PVA loop body. The cs-store can only save 2 cycles of overhead, instead of 3, per iteration. Thus

C_{PVA} = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R3) + 1(exit) = 8.

PVA saving = $(O_l + O_s) * (n_1 + n_2) + (3*n_1 + 2*n_2) - C_{PVA} * n_2$

= (0 + 2) * (704+1) + (3 * 704 + 2 * 1) - 8 * 1 = 3516 cycles

For CVA executions, an additional "mov" function is also introduced. Thus O_s=2. Since t_p=1 (it takes the P unit one cycle to perform a "mov" function) and t_s=0 (there is no secondary arithmetic to perform in the CVA executions), CVA saving = PVA saving = 3517.

Example 6.4:

Vectorized loop shown in Example 1.1 on page 9 using PVA and CVA constructs. Estimate the execution cycles for the vectorized loop, assuming that the vector length is 100.

Table 6.8: Vectorizing A Loop For Performing C[i] = A[i] * B[i]

Assembly Instructions	Using PVA Construct Only	Using CVA Construct Only

addi r2,2
ld.h r7,(r2)
addi r3,1
ld.b r6,(r3)
mul r7,r6 (2 cycles)
st.h r7,(r2)
decne r1
bt L1
	PVA @L0,@L1,@S, #1
	cs-store:
	mul r0, r1, @S
		CVA mul @L0,@L1,@S;

The scalar loop takes 13 * 100 + 12 * 1 = 1312 cycles. In this loop, $O_l + O_s = 8$, $O_{lp} = 3$, $t_s=0$, $t_p=2$ (a 2-cycle multiplication is performed at the P unit).

C_{PVA} = 3(SSR) + 1(CIR) + 2(inst. decode) + 3(R0/R1/R2) + 1(exit) = 10 cycles.

PVA saving = $(O_l + O_s) * (n_1 + n_2) + (3*n_1 + 2*n_2) - C_{PVA} * n_2$

= 8 * 100 + (3*99 + 2*1) - 10 * 1 = 1089 cycles, or a speedup of 5.88.

$C_{CVA} = C_{PVA} + t_p + t_s - 1$ = 10 + 2 + 0 - 1 = 11 cycles.

CVA saving = $(O_l+O_s) * (n_1+n_2) + (t_p+t_s+2)*n_1 + 2*n_2 - C_{PVA}*n_2$

= 8 * 100 + (2 + 0 + 2)*99 + 2*1 - 11*1 = 1187, or a speedup of 10.49.

In this example, PVA executions achieve a speedup of 5.88; while the CVA executions achieve a speedup of 10.49. This disparity is mainly due to the pipelining of the 2-cycle multiply function at the P unit. For PVA executions, the machine is unable to simultaneously execute the multiply functions from different iterations. That is, the machine can initiate a new iteration only when the current iteration is completed. It does not possess the knowledge that the two multiplication functions across different iterations are actually independent operations.

However, if the multiply at the P unit takes only one cycle to execute, then CVA and PVA would have achieved the same performance. □

6.7 CVA-Only vs. PVA-Only vs. CVA/PVA Executions

When a loop is vectorizable with a CVA construct, the vectorized loop typically performs equally or better than the PVA version of the loop. Some loops, however, are not vectorizable

using the CVA construct. In these cases, PVA provides an opportunity for improving the performance.

However, there are also some loops that perform the best when a combination of both constructs are used. The following is an example of such loops.

Example 6.5:

Vectorize the following using (i) PVA-only; (ii) CVA-only; and (iii) CVA/PVA constructs. Estimate their speedups assuming that the loop is executed 100 times.

```
L1:
subi      r13,1
ldb       r6,(r13,1)
lsli      r6,1
rsub      r6,r11
add       r6,r7
ldw       r7,(r12)
decne     r10
sth       r7,(r6) //not constant-stride
addi      r12,4
bt        L1
```

This loop is taken from the benchmark program called "`jpeg`". It contains a non-constant stride store operation; it is thus not CVA vectorizable. The scalar loop takes 14 x 99 + 13 x 1 = 1399 cycles to execute.

The PVA-only version of the loop is shown in second left most column of Table 6.9. The PVA loop removes the overheads for the two cs-load instructions and the loop control mechanism.

This Table also shows how the loop can be vectorized using a combination of a CVA instruction and a PVA instruction (shown in the right most column of the Table). The former instruction is a compound CVA. It reads in a vector and performs two ALU functions ("lsli" and "rsub") with a throughput rate of one (not including the vector setup/exit costs). This CVA instruction then writes its output to a temporary vector.

The PVA instruction then takes over the remaining tasks of the loop: it reads back in the temporary vector, calculates the store addresses and performs the non-constant stride store operations.

Table 6.9: A Critical Loop From "jpeg"

Scalar Code	(i) Using PVA-only Construct	(ii) Using CVA-only Construct	(iii) Using CVA/PVA Constructs

L1:
subi r13,1
ldb r6,(r13,1)
lsli r6,1
rsub r6,r11
add r6,r7
ldw r7,(r12)
decne r10
sth r7,(r6)
addi r12,4
bt L1
	PVA @L0,@L1,#6
	mov r6, r0
	lsli r6,1
	rsub r6,r11
	add r6,r7
	sth r1,(r6)
		Not vectorizable.	<some initialization code>
			; read a vector via L0 and write
			; to a temporary vector via S
			CVA lsli @L0, 1, @P,
			rsub @P, r11, @S;
			.....
			<some initialization code>
			; read from the temporary
			; vector via L0
			PVA @L0, #3
			mov r6, r0
			add r6, r7
			sth r1, (r6)

PVA-only Executions

For PVA-only executions, O_l=5, O_s=0, O_{lp} = 3. O_s is 5 cycles instead of 6 because an additional "mov r6,r0" instruction is introduced in the PVA loop body. This is because register R0, which is the head of the L_0 stream, is a read only register. The two-operand M-CORE ISA destructs one of its source operands when performing many ALU function ("lsli" in the above example). The "mov" instruction is used to ensure that we don't write back to R0. Thus,

C_{PVA} = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit) = 9

PVA saving = $(O_l + O_s) * (n_1 + n_2) + (3*n_1 + 2*n_2) - C_{PVA} * n_2$

= (5 + 0) * (99 +1) + (3 * 99 + 2 * 1) - 9 * 1 = 790 cycles

Speedup for PVA-only executions = 1399/(1399-790) = 2.297

CVA/PVA Executions

For CVA/PVA executions, lets consider first, the CVA instruction. In this case, t_p=t_s=1.

C_{CVA} = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + t_p + t_s - 1 = 10

CVA execution time = C_{CVA} + 100 = 110 cycles

For the PVA instruction,

C_{PVA} = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8

PVA execution time = C_{PVA} + 4 * 100 = 408 cycles

Total execution time = 110 + 408 = 518 cycles.

Speedup for CVA/PVA executions = 1399/518 = 2.700

Thus the CVA/PVA executions achieve a higher speedup (2.700) compared to those achieved by the PVA-only executions (2.297) and by CVA-only executions (not vectorizable). □

6.8 TM Strip-Mining Costs

For CVA executions, when a temporary register allocated into TM is larger than the size of TM, the latter will need to be strip-mined (see Section 4.8.4 on page 70). In this Section, we will estimate the execution costs associated with strip-miming TM.

The following shows a strip-mined code in M-CORE instructions. The corresponding high level c-code is shown in Example 4.10 on page 70. In this code, the operand size for the vector operations is assumed to be a word (4 bytes); TM (512 bytes) can thus store 128 elements.

The code contains a loop with 2-level of nesting. The outer loop executes n/128+1 iterations. The inner loop is invoked n/128+1 times; it executes (n mod 128) iterations during the first invocation, and 128 iterations for each of the subsequent n/128 invocations.

```
//-------------------------------------------------------------
// Strip-mined code for TM (see Example 4.10 on page 70).
// ===========================================================
//
// Memory Variables:
//   LOW       - low
//   VL        - VL
//   N_DIV_128 - n/128
//   J_INDEX   - j loop index for outer loop
// Registers:
//   r14       - i loop index for inner loop
//   r13       - low+VL-1
//   r0        - initially contains n (vector length)
//-------------------------------------------------------------

//---------------
//    Prolog
//---------------
movi   r1, 1
lrw    r2, [LOW]
stw    r1, (r2)          // low = 1

movi   r1, 0x3f
and    r1, r0
lrw    r3, [VL]
stw    r1, (r3)          // VL = n mod 128

mov    r1, r0
asri   r1, 7
lrw    r3, [N_DIV_128]
stw    r1, (r3)          // N_DIV_128 = n/128

movi   r1, 0
lrw    r3, [J_INDEX]
stw    r1, (r3)          // j = 0
cmplt  r1, r0            // j < n/128?
bf     EXIT

//---------------
// Outer Loop
//---------------
// Register allocations:
// r14 = i
```

```
// r13 = low+VL-1

OUTER_LOOP:
lrw    r2, [LOW]
ldw    r14,(r2)          // r14 = low
lrw    r2, [VL]
ldw    r13, (r2)         // r13 = VL
add    r13, r14
subi   r13, 1            // r13 = low+VL-1

//----------------
// Inner Loop
//----------------
//There is always at least one
// iteration for the INNER_LOOP.

INNER_LOOP:

      [Main CVA/PVA executions]

addi   r14, 1            // i = i + 1
cmplt  r14, r13          // (i<low+VL-1)?
bt INNER_LOOP

//------------------
// End of Inner Loop
//------------------
lrw    r2, [LOW]
ldw    r1, (r2)          // r1 = low
lrw    r4, [VL]
ldw    r3, (r4)          // r3 = VL
add    r1, r3
stw    r1, (r2)          // low = low + VL
movi   r3, 127
addi   r3, 1
stw    r3, (r4)          // VL = 128

lrw    r2, [J_INDEX]
ldw    r1, (r2)         // r1 = j
addi   r1, 1
stw    r1, (r2)         // j = j + 1
lrw    r2, [N_DIV_128]
ldw    r2, (r2)         // r2 = n/128
cmplt  r1, r2           // (j<n/128)?
bt     OUTER_LOOP

//------------------
// End of Outer Loop
//------------------
EXIT:
   . . . .
.data
.align word
LOW:
.long 0
N_DIV_128:
.long 0
J_INDEX:
.long 0
```

The execution time of this code can be broken down into: execution time for the prolog code (X), setup cost for the inner loop (Y_1), the loop control overhead for the outer loop (Y_2), the loop

control overhead for the outer loop (Z), and the actual CVA execution time itself. These costs are illustrated in Figure 6.2.

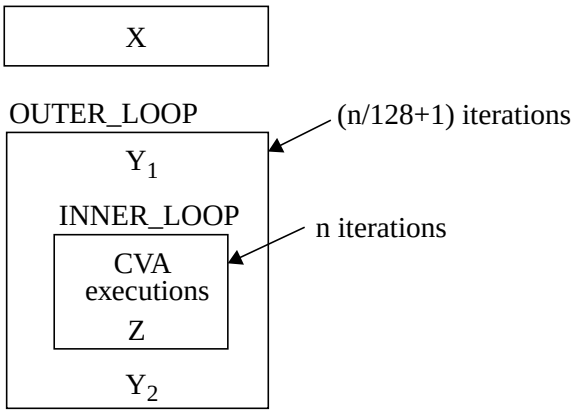


Figure 6.2: Execution Costs For TM Strip-Mined Code

Excluding the inner loop executions, the execution cost is given by: X + (n/128+1) * (Y_1 + Y_2). The inner loop execution cost is given by: (n/128+1) * (C_{CVA} + Z) + n, where C_{CVA} is the setup and exit costs for the CVA executions as described in Section 6.4 on page 91. From the above strip-mined assembly code, X=24, Y_1=10, Y_2=29, Z=4.

The total execution time for the strip-mined code is thus: 24 + (n/128+1) * (43 + C_{CVA}) + n.

In a more general setting,

Execution time for trip-mined code = 24 + (n/m+1) * (43 + C_{CVA}) + n (6.5)

where n is the original vector length;

m is the vector length of strip-mined vectors reside in TM; and

"/" denotes the integer "divide" rounded down to the nearest integer.

Example 6.6:

Calculate the performance improvements and speedups when vectorizing the program loop shown in Example 4.4 on page 53 with CVA constructs.

To recap, this loop, taken from benchmark "`blit`", performs a vector operation described by C[i] = (lsr(A[i], r9) | lsl(A[i], r8)). The vectorized code is shown in Table 6.10. The profile statistics of this loop are shown in Table 6.11.

This loop was invoked three times, with 1000 iterations per invocation (vector length, n=1000). The original scalar loop takes 14 x 2997 + 13 x 3 = 41997 cycles to execute.

Table 6.10: Vertorizing Critical Loop 1 From Benchmark "`blit`"

Address Opcode //; Assembly Code	Using CVA Construct

00000304 8a0e //; ldw r10,(r14)
00000306 25f4 //; decne r4
00000308 12a7 //; mov r7,r10
0000030a 0b97 //; lsr r7,r9
0000030c 1e37 //; or r7,r3
0000030e 12a3 //; mov r3,r10
00000310 970d //; stw r7,(r13)
00000312 1b13 //; lsl r3,r1
00000314 203e //; addi r14,4
00000316 203d //; addi r13,4
00000318 e7f4 //; bt 0x0000304
	Setup a temporary vector T
	. . . . . .
	; T[i] = lsr(A[i], r9)
	mov r5, r9
	CVA lsr @L0, r5, @S;
	. . . . . .
	;@L1 <---- T[i]
	mov r5, r1
	CVA lsl @L0, r5, @P,
	or @L1, @P, @S;

Table 6.11: Profile For Critical Loop 1 From Benchmark "`blit`"

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000304 00000318	target bt	3000 3000	- 00000304	- 2997 (99.9)	- 3 (0.100)

TM is used to store a temporary vector produced by the first CVA instruction. Since n>128, strip-mining for TM is necessary. In the strip-mined code, the inner-most loop executes (n/128+1)=8 iterations, each time it is invoked. This inner-most loop consists of two CVA instructions; they are shown in Table 6.10.

In both CVA instructions, an additional "mov" instruction is needed to initialize R5. For the first CVA instruction (t_p=1, t_s=0), the vector setup/exit cost per invocation is 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 1(t_p) + 0(t_s) - 1 + 1(extra "move" inst.) = 10 cycles.

For the second CVA instruction (t_p=1, t_s=1), it is 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) + 1(t_p) + 1(t_s) - 1 + 1(extra "move" inst.) = 12 cycles.

Thus the total vector setup/exit cost per invocation, C_{CVA} = 10 + 12 = 22 cycles; the execution time per invocation = 24 + (n/128+1) * (43 + C_{CVA}) + 2*n = 2544

For 3 invocations, the saving for CVA-only executions is 41997 - 3 * 2544 = 34365, or a speedup of 5.503. □

6.9 Throughput Rates For CVA Executions With Memory Conflicts

For CVA executions, after the initial pipeline warm-up cost is incurred, a result is produced in each subsequent cycle, provided that there is no memory conflict between the input and output

data streams at M0 and M1; and between the two input data streams at TM. This throughput rate of *one* is the best possible throughput rate on this machine (not including the initial vector setup/ exit costs).

From the right most column of Table 4.3 on page 68, we can see that when there is a conflict at either M0, M1 or TM, only up to a maximum of two data streams are involved in the conflict. In this case, accessing to the memory module in question are alternatively granted to the two contending data streams. As a result, the throughput rate is degraded to one vector result in every *two* cycles (not including the initial vector setup/exit costs).

Example 6.7:

Calculate the performance improvements and speedups for Example 6.6, *assuming that the TM is not used (or TM is removed from the design)*.

For CVA executions, the first CVA instruction are not affected by the absent of TM. It streams in a vector from M0 (or M1) and streams out to M1 (or M0). For the second CVA instruction, however, the throughput rate is reduced to one result every 2 cycles due to conflicts at M0 or M1 (depending on the destination of stream S); an additional 1000 (vector length, n=1000) execution cycles is required per invocation.

With C_{CVA}=22, the execution time per invocation = C_{CVA} + n (for the first CVA instruction) + 2*n (for the second CVA instruction) = C_{CVA} + 3*n = 3022.

For 3 invocations, the saving for CVA-only executions is 41997 - 3 * 3022 = 32931, or a speedup of 4.632.. □

6.10 Maximizing The Use Of TM via Vector Duplication

Certain vector operations could only benefit (performance-wise) from the use of TM by re-allocating some *non-temporary* vectors into the TM. Consider the following vector operation.

B[i] = s *A[i]+B[i], for all i and some scalar s.

This vector operation requires three memory accesses to produce each result element (two reads and one write). In this case, however, source vector B, is also the destination vector. Thus accessing vector B will cause a read and a write conflict in each cycle, regardless of which mem-

ory bank vector B resides. Without any code modification, the performance of this vector execution will suffer.

In this machine, TM is tightly coupled into the execution datapath; the IO system has no access to TM. In this example, if vector B needs not be accessed by the IO system, then the above vector operation can be replaced by the following two *alternatively* invoked vector operations.

$B_1[i] = s * A[i] + B_2[i]$

$B_2[i] = s * A[i] + B_1[i]$

That is, vector B now has two instances, B_1 and B_2. At any given point in time, only one instance is valid. All other references to vector B in the original code will need to be replaced by references to the valid instance of vector B, at that point in time. Figure 6.3 shows how the vector operations and the references to vector B are being transformed in order to take advantage of TM.

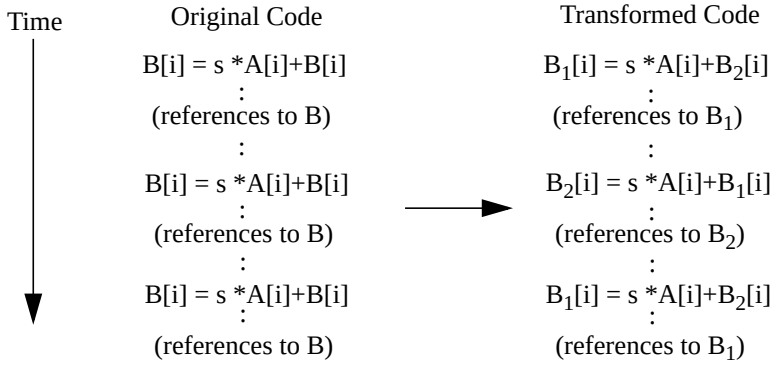


Figure 6.3: Vector Duplication

With these code transformations, an instance of vector B can reside in TM, while the other one in M0 or M1. The vector operations can now utilize TM to improve performance.

6.10.1 Software Implementation of Vector Duplication

To perform vector duplication described above, a global pointer to vector B is needed. Figure 6.4 shows how the updating of this pointer can be implemented. In this figure, vector B has two possible locations, addr_X and addr_Y. In the vector operation, $B_1[i]=A[i]*B_2[i]$, vector B_1 is pointed to by pointer_B1 and vector B_2 by pointer_B2. pointer_B1, the global pointer, always points to the valid instance of B.

```
If (pointer_B1 == addr_X) {
      pointer_B1 = addr_Y;
      pointer_B2 = addr_X;
} else {
      pointer_B1 = addr_X;
      pointer_B2 = addr_Y;
}
< vector operations for B1[i] = A[i] * B2[i] >

// In the rest of the program, pointer_B1
// always points to the valid instance of B.
```

Figure 6.4: Transformed Code For Vector Duplication

6.10.2 Execution Overheads of Vector Duplication

The overhead for *each* invocation of the vector execution includes: a global pointer fetch (2 cycles); a test instruction (1 cycle); two pointer update instructions (2 cycles); and a conditional branch (2 cycles); a total of 7 cycles.

6.11 Instruction Fetch Bandwidth

As mentioned in Section 1.5 on page 9, a subtle benefit of vector processing is the reduced instruction fetch bandwidth. In this Section, we will illustrate how we could estimate the instruction fetch bandwidth reduction associated with the PVA and CVA executions.

We distinguish the difference between instruction requests (IReq) from the processor core and the instruction fetches (IFetch) from the memory M0. This different is illustrated in Figure 6.5.

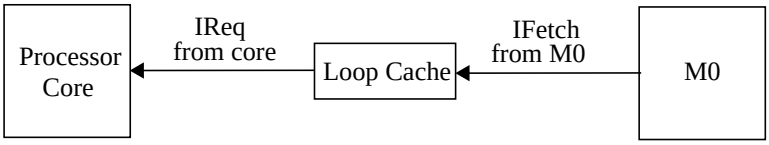


Figure 6.5: IReq from Processor Core versus IFetch from Memory M0

For CVA executions, there is no different between these two metrices, since CVA executions do not utilize the loop cache. For PVA executions, the loop cache satisfied some of the instruction requests from the processor core. As a result, the IFetch from the memory are less than the IReq made by the core.

We define the *Normalized Instruction Request* (*IReq*) from the processor core, as the ratio between the number of instruction requests being made by the processor core for the vectorized program, versus those for the base machine.

Likewise, we define the *Normalized Instruction Fetch* (*IFetch*) from the processor core, as the ratio between the number of instruction fetches being made from the memory M0 for the vectorized program, versus those for the base machine.

Example 6.8:

Calculate the normalized IReq and normalized IFetch for the loop shown in Example 4.4 on page 53, by using (i) a PVA construct; and (ii) a CVA construct. This loop is assumed to execute 100 iterations.

The PVA and CVA versions of the loop are shown in Table 6.12.

Table 6.12: Vectorizing The Loop Shown in Example 1.1

Original Scalar Loop	Using PVA Construct	Using CVA Construct

L1:
ldw r10,(r14)
decne r4
mov r7,r10
lsr r7,r9
or r7,r3
mov r3,r10
stw r7,(r13)
lsl r3,r8
addi r14,4
addi r13,4
bt L1
	<Some vector setup code>
	PVA @L0,@S, #5
	mov r3, r0
	mov r7, r3
	lsr r7,r9
	lsl r3,r1
	cs-store:
	or r3,r7[a]
		<Some vector setup code>
		mov r5, r9
		CVA lsr @L0, r5, @S;
		
		<Some vector setup code>
		mov r5, r8
		CVA lsl @L0, r5, @P,
		or @L1, @P, @S;

There are 11 instructions in the original scalar loop. In this machine, each instruction request fetches two 16-bit instructions. Due to the pipeline nature of the scalar machine, when the branch instruction “bt” is taken, one additional instruction (the fall-through instruction following it) is also fetched. As a result, 12 instructions are being fetched per loop iteration, even though we only execute 11 of them. During the last iteration when the “bt” instruction is not taken, the fall-through instruction is fetched and executed. This instruction is not considered as part of the loop executions. Thus the IReq of the scalar machine during loop executions is given by 12 x 99 + 11 x 1 = 1199 instructions.

For PVA execution, the vector setup code consists of initialization of the following registers:

SSR (2 instructions); CIR (1 instruction); R0 and R2 (1 instruction each). In addition, the PVA instruction is 32-bit wide and is counted as 2 instructions. There are 5 instructions in the loop body, which is executed 100 times. Thus for PVA execution, the IReq is (2+1+1+1+2) + 5 x 100 = 507 instructions

For CVA execution, there are two CVA instructions. The vector setup code for the first CVA instruction consists of initialization of the following registers: SSR (2 instructions); CIR (1 instruction); R0 and R2 (1 instruction each). In addition, there is also an additional “mov r5, r9” instruction. The CVA instruction is also counted as 2 instructions. The total IReq for the initialization code is 7 instructions.

The vector setup code for the second CVA instruction consists of initialization of the following registers: SSR (2 instructions); CIR (1 instruction); R0, R1 and R2 (1 instruction each). In addition, there is also an additional “mov r5, r8” instruction. Again, the CVA instruction is counted as 2 instructions. The total IReq for the initialization code is 8 instructions. Thus for CVA executions, IReq is given by 7 (first CVA instruction) + 8 (second CVA instruction) 15 instructions

The normalized IReq for PVA and CVA executions are summarized in Table 6.13.

Table 6.13: Normalized IReq For PVA and CVA Executions

Execution Modes	PVA Execution	CVA Execution
IFetch	507 instructions	15 instructions
Normalized IFetch	0.4229	0.0125

In this example, the PVA execution is able to reduce the instruction fetch bandwidth from the original 1199 instructions down to 507 instructions - a reduction of about 58%; the CVA execution is able to further reduce the instruction bandwidth down to 15 instructions - a reduction of about 99%.

For PVA execution, the reductions are primarily due to eliminating the fetching of instructions that perform the cs-load, cs-store and loop control operations. Once these operations are prespecified in the vector instruction, there is no need for the machine to fetch these instructions again due to their repetitive nature.

The CVA execution goes one step further in reducing the instruction fetch bandwidth. Once all the data streams, primary and secondary arithmetic are properly setup, there is no need to fetch any instruction for the rest of the CVA executions. □

CHAPTER 7

EXPERIMENTAL RESULTS

In this Chapter, we will present the performance results for the pseudo-vector machine. We will first present the overall speedups, as well as the speedups during program loop executions. We will then present the performance impact by varying the sizes of TM.

For PVA-only executions, we will present what are the performance improvements contributed by removing various types of overheads associated with loop executions. Later in this Chapter, we will also present what are the effects of vectorizations on instruction fetch bandwidth on this machine.

7.1 Overall Speedups

Figure 7.1 and Table 7.1 show, for each benchmark, the overall speedups for various execution modes on this machine. These results assume that the TM is 512 bytes. Depending on the way we restrict the selection of the vectorizing constructs, there are three major execution modes: (i) PVA-only executions; (ii) CVA-only executions; and (iii) CVA/PVA executions (see Section 6.3 on page 90).

In CVA-only executions, two different sets of results are presented. The first set of results, called the “CVA-only executions without early termination”, corresponds to the cases where we would *not* vectorize a loop with a CVA construct if it contains an early terminating condition. An example of such a loop is shown in Example 3.3 on page 46. In the second set of results, called the “CVA-only executions with early termination”, we allowed a vector computation to exit early. An example of such vectorized loop is shown in Example 3.4 on page 45. This latter approach is more generic and provides more opportunities for vectorization.

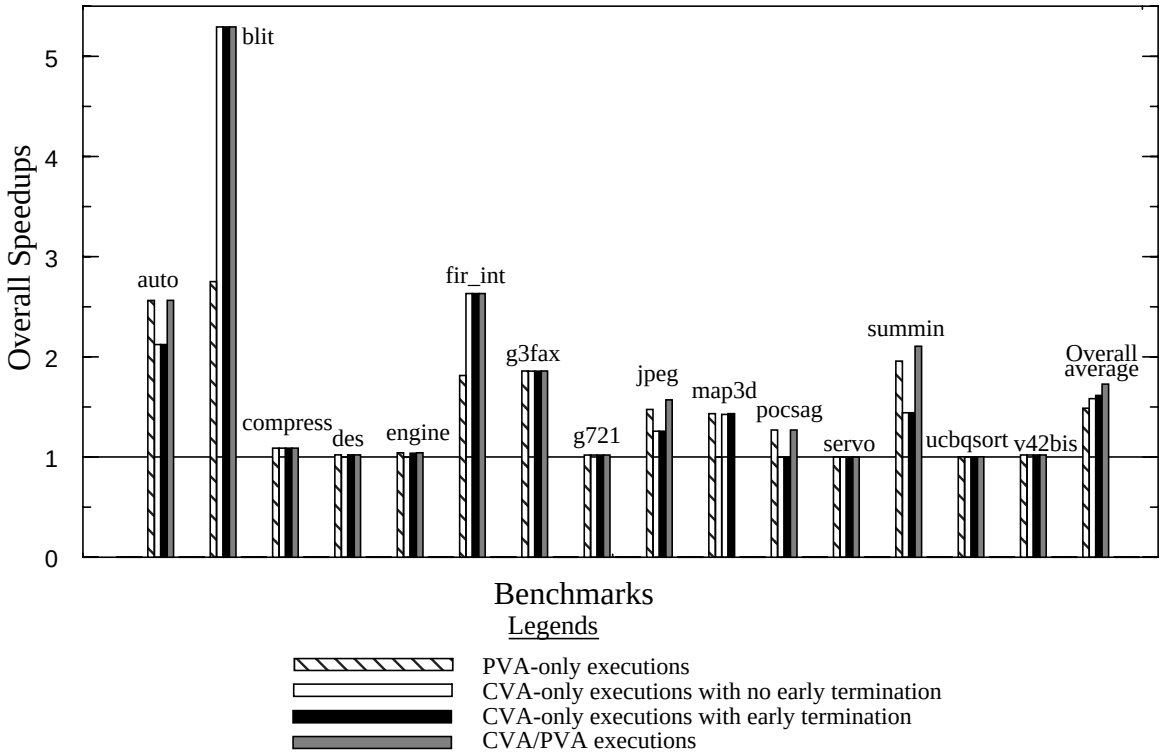


Figure 7.1: Overall Speedups For Various Execution Modes

Table 7.1: Overall Speedups For Various Combinations of Execution Modes[1]

Benchmarks	PVA-Only Executions	CVA-Only Executions		CVA/PVA Executions
		Without early termination	With early termination	
auto	2.563	2.123	2.123	2.564
blit	2.751	5.293	5.293	5.293
compress	1.089	1.089	1.089	1.089
des	1.022	1.000	1.022	1.022
engine	1.043	1.000	1.038	1.043
fir_int	1.814	2.632	2.632	2.632
g3fax	1.859	1.858	1.858	1.859
g721	1.020	1.020	1.020	1.020
jpeg	1.475	1.256	1.256	1.569
map3d	1.433	1.000	1.426	1.433
pocsag	1.270	1.000	1.000	1.270
servo	1.000	1.000	1.000	1.000
summin	1.958	1.480	1.480	2.190
ucbqsort	1.002	1.002	1.002	1.002
v42bis	1.021	1.021	1.021	1.021
Average	**1.488**	**1.585**	**1.617**	**1.734**

1. TM is assumed to be 512 bytes.

If we allow both CVA and PVA executions with *all* the vector execution capabilities described in this dissertation, the overall average speedup achieved is 1.734 - the highest among all possible execution modes. Closely following this, is the CVA-only executions with early termination, with an overall speedup of 1.617. By not allowing the vector executions to exit early, the overall speedup drops slightly, to 1.585. Using the PVA-only executions, we achieved an overall speedup of 1.488.

7.1.1 CVA-Only vs. PVA-Only Executions

As mentioned earlier, if a loop is CVA-vectorizable, it is also PVA-vectorizable. The PVA construct represents a more generic vectorizing mechanism. It thus provides more opportunities for vectorization and performance improvements.

However, if a program loop is vectorizable by either a CVA or a PVA construct, depending on the type of primary arithmetic (p_op) performed at the P unit, a CVA-vectorized code can typically achieve a higher speedup than its PVA counterpart.

If p_op is a multi-cycle operation (such as a 2-cycle multiply or divide), then the CVA executions will achieve higher speedup. This is because for PVA executions, the machine is unable to simultaneously execute the multi-cycle functions across different iterations. That is, it can initiate a new iteration only when the current iteration is completed. This scenario was illustrated in Example 6.4 on page 100.

In general, if a loop is vectorizable either by a CVA or a PVA construct, the former typically performs the same or better than the latter. The CVA construct will outperform the PVA construct when:

- p_op takes multiple cycles to execute; or
- the CVA is a reduction or hybrid CVA; or
- the CVA is a compound CVA that also utilizes the s_op function, such as the vector operation described by $C[i] = A[i]^2 + B[i]$.

To perform $C[i] = A[i]^2 + B[i]$, in every cycle, the compound CVA reads two data from the memory, performs a “multiply” and an “add” operations, writes one data to the memory - all in a single cycle.

7.1.2 Allowing CVA-Only Executions To Terminate Early

In CVA-only executions, if we allow a vector arithmetic to terminate early, we were able to

vectorize more program loops using the CVA construct. By incorporating this capability, the overall speedup increases slightly from 1.585 to 1.617.

For CVA-only executions, only three benchmark programs benefited from this early termination capability. Out of these three benchmarks, `map3d` benefited the most from this capability. The speedup, in this cases, increased from 1.000 (no speedup) to 1.426.

7.2 Speedups During Program Loop Executions

Figure 7.2 and Table 7.2 show the speedups during program loop executions for the various execution modes. Again, TM is assumed to 512 bytes.

Table 7.2: Speedups During Loop Executions For Various Combinations of Execution Modes[1]

Benchmarks	PVA-Only Executions	CVA-Only Executions		CVA/PVA Executions
		Without early termination	With early termination	
auto	6.671	3.806	3.806	6.676
blit	2.794	5.503	5.503	5.503
compress	5.994	5.994	5.994	5.994
des	5.929	1.010	5.929	5.929
engine	1.740	1.000	1.601	1.740
fir_int	2.805	9.015	9.015	9.015
g3fax	5.500	5.499	5.499	5.500
g721	3.143	3.143	3.143	3.143
jpeg	3.124	6.696	6.696	4.258
map3d	4.930	1.000	4.711	4.930
pocsag	1.531	1.000	1.000	1.531
servo	1.000	1.000	1.000	1.000
summin	2.501	7.645	7.645	2.998
ucbqsort	1.726	1.726	1.726	1.726
v42bis	5.993	5.991	5.991	5.991
Average	**3.692**	**4.002**	**4.617**	**4.396**

1. TM is assumed to be 512 bytes.

If we consider the speedup only during program loop executions, the speedups achieved were much higher, as can be expected. In these cases, the CVA-only executions with early termination achieves an average speedup of 4.617 - the highest among all execution modes. For PVA-only executions, the average speedup during loop executions is 3.692. For CVA/PVA executions,

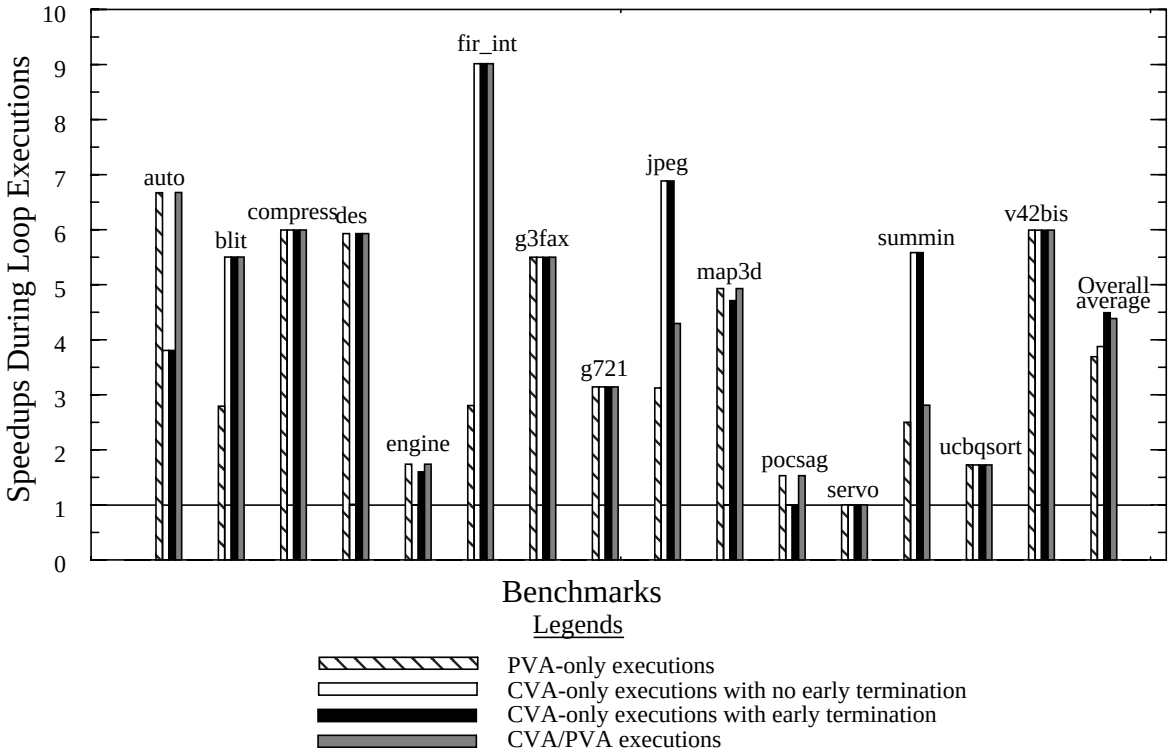


Figure 7.2: Speedups During Loop Executions For Various Execution Modes

the corresponding speedup is 4.396.

7.3 Performance Impact By Varying The Sizes Of TM

In this machine, TM is primarily used to increase the effective memory bandwidth for CVA executions - it is essentially used as a third memory port in addition to M0 and M1. A compound CVA that enables all three data streams (L_0, L_1 and S) could utilize TM to improve the performance. Without TM, such compound CVA would have taken two cycles to produce a result, instead of one (see Section 6.9 on page 107).

In this work, the PVA-only executions do not utilize TM, and thus its performance was not affected. For CVA executions, unfortunately, there were only three benchmarks (namely, `blit`, `jpeg` and `summin`) whose performance could benefit from using TM; all other benchmarks did not benefit, in terms of performance, from using TM.*

The low utilization of TM is due to the fact that most of the benchmarks did not have three-data stream CVA in their vectorized code. Since all vectorizations were manually done by hand (by examining the assembly code), we believe that more opportunities could have been exposed or

* TM could also be used by these benchmarks for power reduction. However, evaluating the power benefits of using TM is beyond the scope of this dissertation.

made available if these were done by the compilers, in conjunction with careful high-level coding. However, no such high-level code modification nor compiler work were done in this work.

Among the abovementioned three benchmarks, `blit` has two critical loops that utilized TM without the need to use the vector duplication approach (see Section 6.10 on page 108). `jpeg` and `summin`, on the other hand, each has one critical loop that utilized TM by using the vector duplication approach.

7.3.1 TM Strip-Mining Costs vs. TM Sizes

The executions of a three-data stream CVA do not always benefit from using TM, due to its strip-mining costs. These costs are higher when the number of strip-mined iterations is large; or equivalently, when TM is small, relative to the total vector length.

Benchmark "`blit`"

Figure 7.5 shows how the speedups for `blit` vary as a function of TM sizes. This figure is for the CVA/PVA executions. `blit` spent about 99% of its executing time on two critical loops, manipulating two vectors of length 1000. Both of these loops benefited greatly from using TM. These loops were shown in Example 4.4 on page 53 and Example 6.6 on page 106.

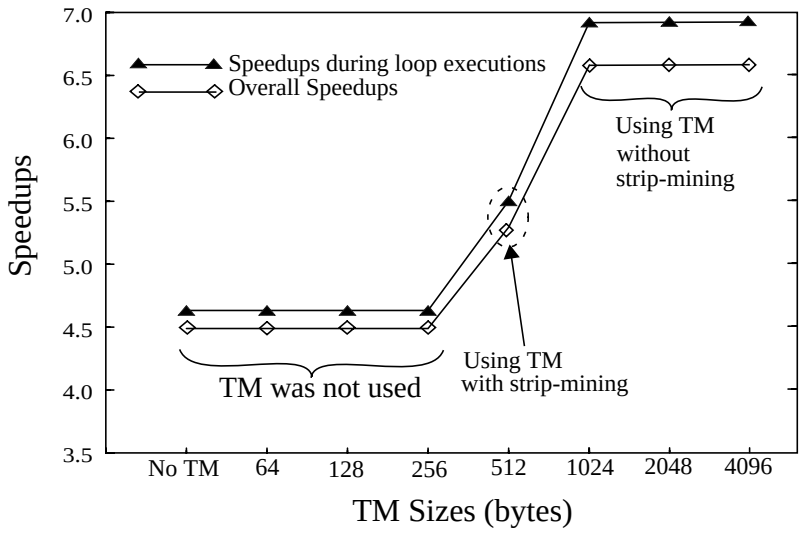


Figure 7.3: Speedups For `blit` vs. TM Sizes

For small TM (256 bytes or less), the costs of strip-mining overwhelmed the performance benefits brought by using the TM itself. Thus for these sizes of TM, the TM was not used. That is, in these cases, the approach of avoiding the use of TM (thereby incurring memory conflicts at M0/M1), still outperformed the approach of using TM with strip-mining.

When TM is 512 bytes in size, the benefits of using TM started to manifest. The overall speedup, in this case, jumps from 4.489 to 5.293. When the size of TM increases to 1024 bytes or larger, the entire temporary vector could fit into TM. As a result, strip-mining of TM was no longer necessary, and the overall speedup increases even further, to 6.581. Increasing TM beyond the size of 1024 bytes does not further improve the performance.

Benchmarks "jpeg" and "`summin`"

For `jpeg` and `summin`, they *each* have one loop that utilized TM for performance. For these benchmarks, the speedups *during the executions of that particular loop* is shown in Figure 7.4. In both cases, the loop could only benefit from TM when TM is 128 bytes or larger. At these sizes, no strip-mining was necessary. The speedups, in these cases, jump from 5.978 to 10.134 for `jpeg`, and from 5.584 to 9.727 for `summin`.

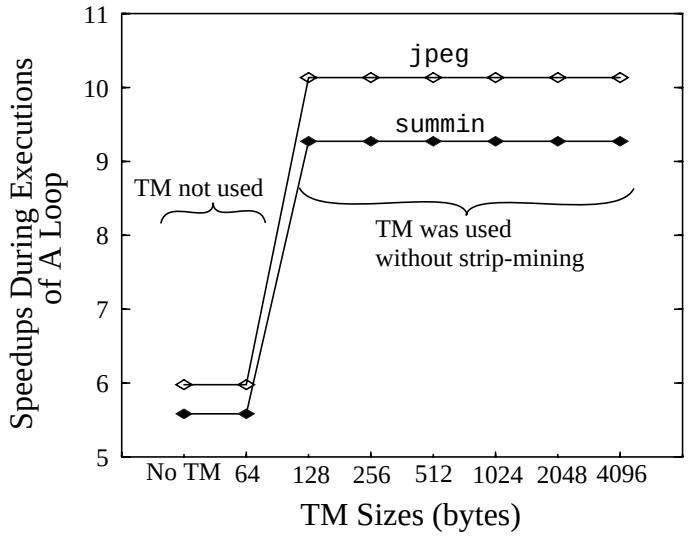


Figure 7.4: Speedups During A Single Loop Execution vs. TM Sizes (For `jpeg` and `summin`)

The overall speedups brought by using TM, for these two benchmarks, are far less impressive. They are shown in Figure 7.5. With TM of 128 bytes or larger, the overall speedup increases from 1.549 to 1.569 for `jpeg`, and from 2.106 to 2.190 for `summin`.

7.3.2 Average Speedups vs. TM Sizes

The speedups versus TM sizes, average over all benchmarks, are shown in Table 7.3, Table 7.4 and Figure 7.6. Table 7.3 is for CVA/PVA executions, while Table 7.4 is for CVA-only executions.

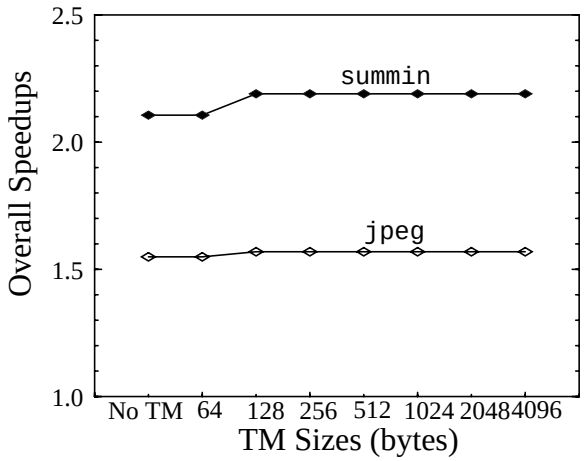


Figure 7.5: Overall Speeds vs. TM Sizes (For `jpeg` and `summin`)

Table 7.3: Speedups For CVA/PVA Executions vs. TM Sizes

Benchmarks	TM Sizes (Bytes)									
	Speedups During Loop Executions					Overall Speedups				
	No TM	128	256	512	1024	No TM	128	256	512	1024
blit	4.632	4.632	4.632	5.503	6.923	4.489	4.489	4.489	5.293	6.581
jpeg	3.971	4.258	4.258	4.258	4.258	1.549	1.569	1.569	1.569	1.569
summin	2.812	2.998	2.998	2.998	2.998	2.106	2.190	2.190	2.190	2.190
Average Over All Benchmarks	**4.306**	**4.338**	**4.338**	**4.396**	**4.490**	**1.673**	**1.680**	**1.680**	**1.734**	**1.819**

Table 7.4: Speedups For CVA-Only[1] Executions vs. TM Sizes

Benchmarks	TM Sizes (Bytes)									
	Speedups During Loop Executions					Overall Speedups				
	No TM	128	256	512	1024	No TM	128	256	512	1024
blit	4.632	4.632	4.632	5.503	6.923	4.489	4.489	4.489	5.293	6.581
jpeg	5.468	6.696	6.696	6.696	6.696	1.244	1.256	1.256	1.256	1.256
summin	5.583	7.645	7.645	7.645	7.645	1.442	1.480	1.480	1.480	1.480
Average Over All Benchmarks	**4.340**	**4.559**	**4.559**	**4.617**	**4.712**	**1.560**	**1.564**	**1.564**	**1.617**	**1.703**

1. With early terminations.

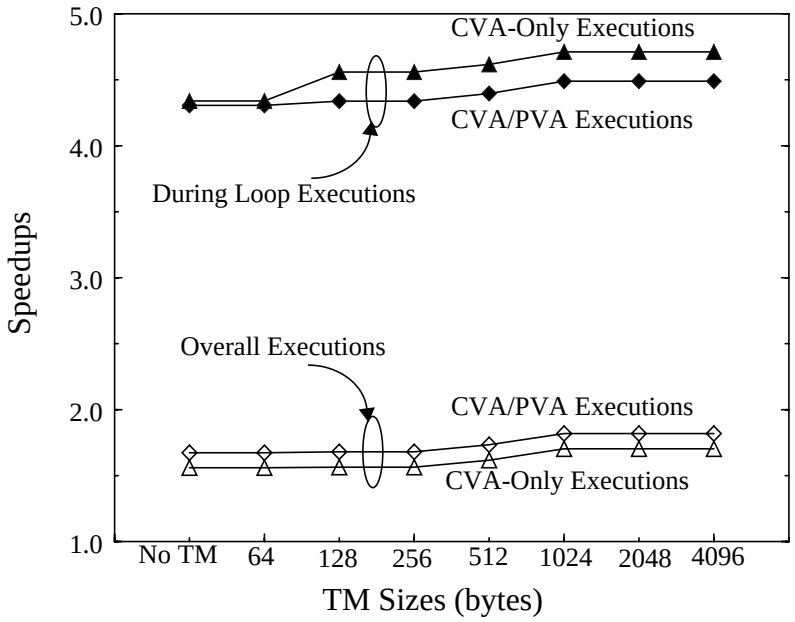


Figure 7.6: Speeds vs. TM Sizes - Average of All Benchmarks

Figure 7.7 highlights the performance benefits of using a 512-byte TM, for both the overall speedups and the speedups during loop executions. The speedups for PVA executions, although not affected by TM, are also shown in this figure for comparison purposes.

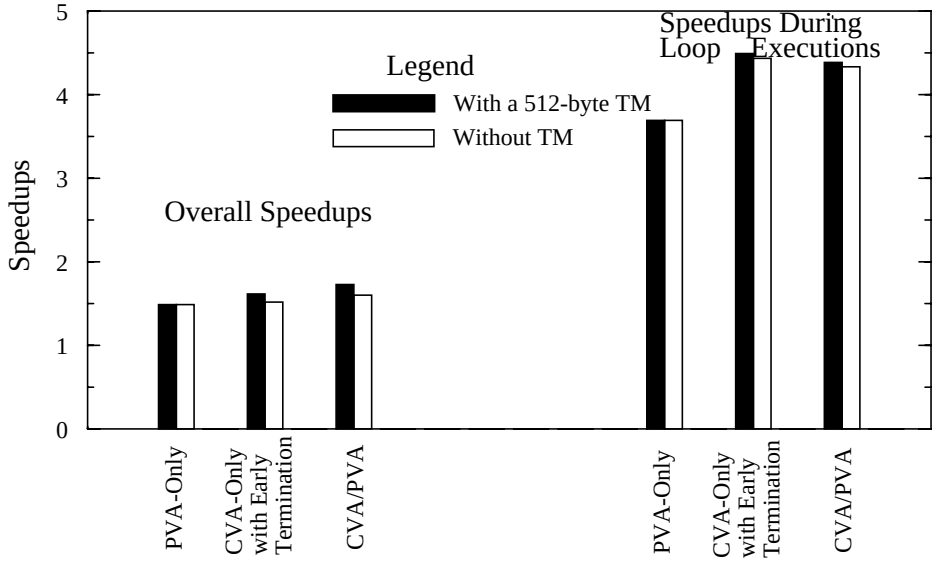


Figure 7.7: Performance Benefits of Using a 512-Byte TM

7.4 PVA-Only Executions

In this Section, we will take a closer look at the performance benefits due to eliminating various types of overheads in PVA-only executions. As described in Section 6.5, these overheads

include lp-ctl-oh, cs-load-oh, cs-store-oh. In order to better quantify the breakdown of the effects of eliminating each of these types of overheads, we will use the performance improvement metric defined in Equation (6.1) on page 87.

Table 7.5 and Figure 7.8 show the performance improvements, for each benchmark, using the PVA-only executions. For each benchmark, three set of results are presented. They represent the performance improvements achieved due to eliminating cs-load-oh, cs-store-oh and lp-ctl-oh, respectively. The sums of these three components give the total improvements for PVA-only executions.

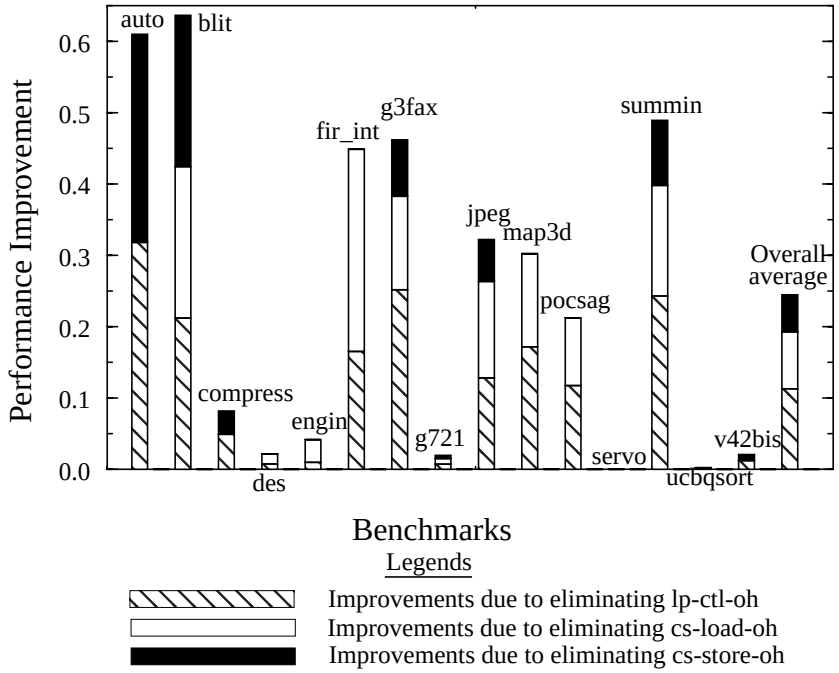


Figure 7.8: Performance Improvements Using PVA-Only Executions

Table 7.5: Performance Improvements For PVA-Only Executions

Benchmarks	Performance Improvements Due To Eliminating Various Types of Overheads			Overall Performance Improvements
	lp-ctl-oh	cs-load	cs-store	
auto	0.3182	0.0000	0.2917	0.6099
blit	0.2121	0.2122	0.2122	0.6365
compress	0.0491	0.0000	0.0327	0.0818
des	0.0075	0.0141	0.0001	0.0217
engine	0.0098	0.0318	0.0000	0.0416
fir_int	0.1652	0.2835	0.0000	0.4487
g3fax	0.2517	0.1311	0.0791	0.4619

Table 7.5: Performance Improvements For PVA-Only Executions

Benchmarks	Performance Improvements Due To Eliminating Various Types of Overheads			Overall Performance Improvements
	lp-ctl-oh	cs-load	cs-store	
g721	0.0072	0.0078	0.0046	0.0196
jpeg	0.1281	0.1352	0.0589	0.3222
map3d	0.1717	0.1306	0.0000	0.3023
pocsag	0.1174	0.0950	0.0000	0.2124
servo	0.0000	0.0000	0.0000	0.0000
summin	0.2430	0.1550	0.0913	0.4893
ucbqsort	0.0008	0.0011	0.0005	0.0024
v42bis	0.0124	0.0000	0.0083	0.0207
Average	**0.1129**	**0.0798**	**0.0520**	**0.2447**

From Table 7.5, eliminating lp-ctl-oh provided the greatest performance improvements (11.29%). This is closely followed by eliminating the cs-load-oh (7.98%). Eliminating the cs-store-oh improves the performance by yet another 5.20%. By removing all these overheads, the overall performance is improved by 24.47%, for PVA-only executions.

7.5 Instruction Fetch Bandwidth Reductions

In this Section, we will look at the effects of vectorizations on IReq from the processor core and the IFetch from the memory (see Section 6.11 on page 110).

7.5.1 Normalized IReq From Processor Core

Table 7.6 shows, for each benchmark, the normalized IReq from processor core for various execution modes.

Table 7.6: Normalized IReq From The Processor Core

Bench-marks	Dynamic IReq. From Core	Normalized IReq. From Processor Core					
		During Loop Executions			Overall		
		PVA-Only	CVA-Only	CVA/PVA	PVA-Only	CVA-Only	CVA/PVA
auto	20695	0.168	2.853×10^{-3}	2.853×10^{-3}	0.3237	0.1893	0.1893
blit	78448	0.4173	1.250×10^{-3}	1.250×10^{-3}	0.4652	0.0834	0.0834
compress	355216	0.2002	1.506×10^{-4}	1.506×10^{-4}	0.9103	0.8878	0.8878
des	519037	0.1804	0.0781	0.0781	0.9799	0.9774	0.9774
engine	1058154	0.5754	0.3633	0.3633	0.9353	0.9030	0.9030
fir_int	705966	0.3577	0.0233	0.0233	0.4248	0.1253	0.1253
g3fax	1681130	0.2069	0.0244	0.0244	0.5495	0.4458	0.4458
g721	256025	0.3584	0.2059	0.2059	0.9830	0.9791	0.9791
jpeg	1528812	0.3267	0.0329	0.1822	0.6802	0.7870	0.6115
map3d	1463233	0.2108	0.0649	0.0649	0.6430	0.5770	0.5770
pocsag	147202	0.6905	1.0000	0.6905	0.8382	1.0000	0.8382
servo	42919	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
summin	1532825	0.4230	0.0777	0.3115	0.4824	0.9000	0.3854
ucbqsort	804662	0.6220	0.4695	0.4695	0.9979	0.9971	0.9971
v42bis	1660493	0.2001	1.45×10^{-4}	1.45×10^{-4}	0.9800	0.9751	0.9751
Average	**-**	**0.3958**	**0.2230**	**0.2638**	**0.7462**	**0.6858**	**0.6650**

Normalized IReq During Loop Executions

Figure 7.9 shows the normalized IReq for various execution modes during loop executions. During loop executions, the PVA-only executions were able to reduce the number of IReq down to 39.58% of those for the base machine. These reductions were primarily due to the fact that many instructions in the original loops specified some cs-load, cs-store and loop control operations. Once these operations are specified in the PVA instructions, the corresponding instructions were no longer needed by the machine during the loop executions.

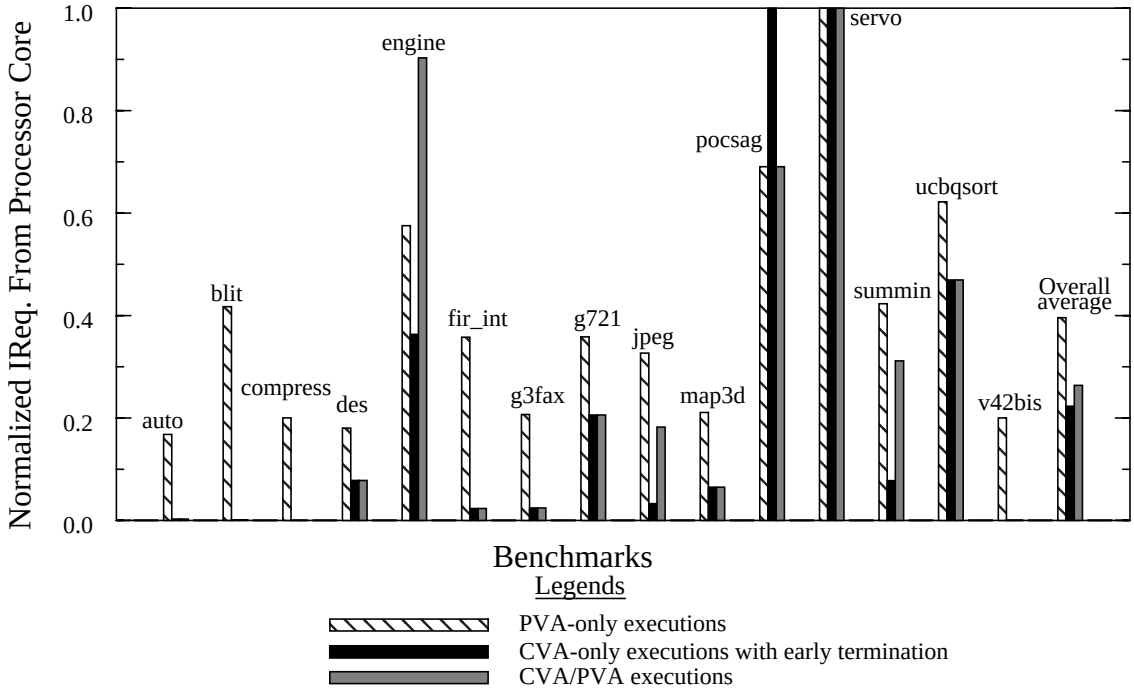


Figure 7.9: Normalized IReq From Processor Core During Loop Executions

The CVA executions were able to reduce the IReq even further, by also pre-specifying all the arithmetic operations (p_op and s_op) in the CVA instructions. During loop executions, the normalized IReq is 0.2230 for CVA-only executions, and 0.2638 for CVA/PVA executions.

Normalized IReq - Overall

Figure 7.10 shows, for each benchmark, the overall normalized IReq from processor core for various executions modes. Overall, the normalized IReq from processor core is 0.7462 for PVA-only executions; 0.6858 for CVA-only executions and 0.6650 for CVA/PVA executions.

7.5.2 Normalized IFetch From The Memory M0

For PVA executions, we distinguish two caching schemes for the loop cache: (i) *essential instruction only caching* (the loop cache only caches all the essential instructions); and (ii) *all instruction caching* (the loop cache can cache all instructions in the loop, including non-essential instructions, as long as the caching of these instructions do not replace any essential instructions). The “essential instruction only caching” represents the approach used by the traditional DSP processors (such as the SHARC 2106x chip). The “all instruction caching” approach is proposed in this dissertation (see Section 5.8 on page 83).

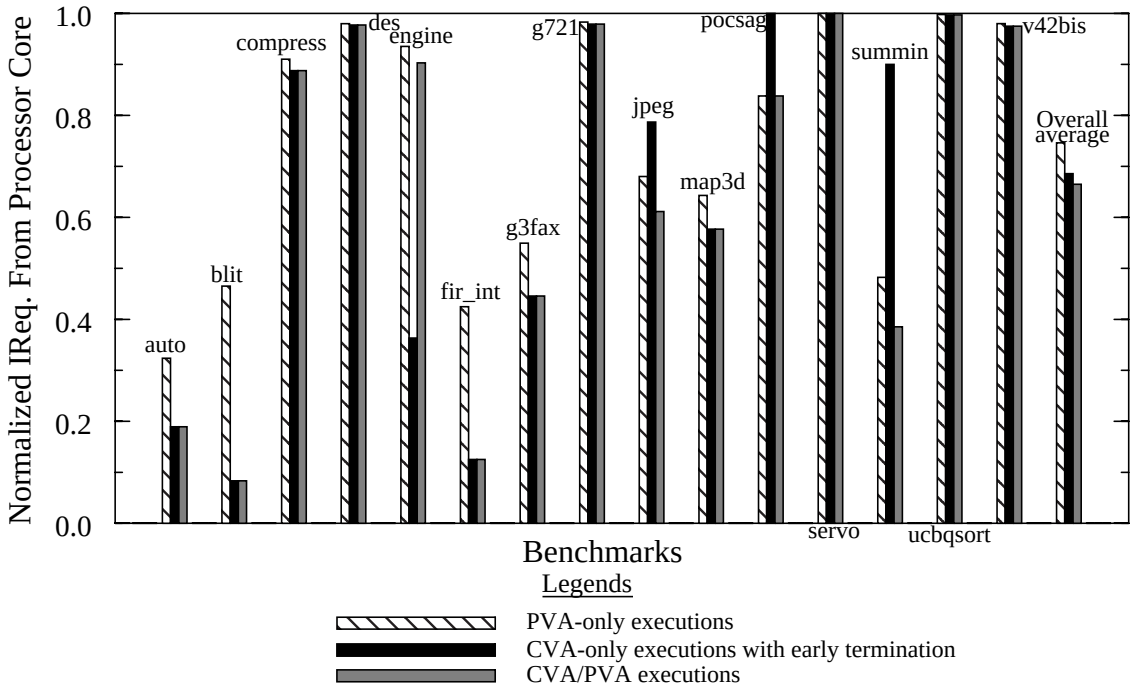


Figure 7.10: Normalized IReq From Processor Core - Overall

Table 7.7: Normalized IFetch From Memory

Bench-marks	During Loop Executions				Overall			
	PVA-Only (essential inst. only caching)	PVA-Only (all inst. caching)	CVA-Only	CVA/PVA	PVA-Only (essential inst. only caching)	PVA-Only (all inst. caching)	CVA-Only	CVA/PVA
auto	$1.545x10^{-3}$	$1.545x10^{-3}$	$2.853x10^{-3}$	$2.853x10^{-3}$	0.1882	0.1882	0.1893	0.1893
blit	0.339	$9.723x10^{-4}$	$1.250x10^{-3}$	$1.250x10^{-3}$	0.3887	0.0832	0.0834	0.0834
compress	$1.757x10^{-4}$	$1.757x10^{-4}$	$1.506x10^{-4}$	$1.506x10^{-4}$	0.8878	0.8878	0.8878	0.8878
des	0.0781	0.0781	0.0781	0.0781	0.9774	0.9774	0.9774	0.9774
engine	0.3633	0.3633	0.3633	0.3633	0.9030	0.9030	0.9030	0.9030
fir_int	0.1347	0.0233	0.0233	0.0233	0.2251	0.1253	0.1253	0.1253
g3fax	0.0244	0.0244	0.0244	0.0244	0.4458	0.4458	0.4458	0.4458
g721	0.2059	0.2059	0.2059	0.2059	0.9791	0.9791	0.9791	0.9791
jpeg	0.0773	0.0260	0.0329	0.0310	0.5617	0.5374	0.7870	0.5397
map3d	0.0649	0.0649	0.0649	0.0649	0.5770	0.5770	0.5770	0.5770
pocsag	0.5721	0.0267	1.0000	0.0267	0.7763	0.4912	1.0000	0.4912
servo	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
summin	0.2116	0.0493	0.0777	0.0474	0.2928	0.1472	0.9000	0.1455
ucbqsort	0.4697	0.4697	0.4695	0.4695	0.9971	0.9971	0.9971	0.9971
v42bis	$1.693x10^{-4}$	$1.693x10^{-4}$	$1.45x10^{-4}$	$1.45x10^{-4}$	0.9751	0.9751	0.9751	0.9751
Average	**0.2362**	**0.1556**	**0.2230**	**0.1560**	**0.6783**	**0.6210**	**0.7218**	**0.6210**

Figure 7.11 shows the normalized IFetch from the memory, for both the overall executions and loop executions. The lower the normalized IFetch from M0, the lower access power will be consumed at M0.

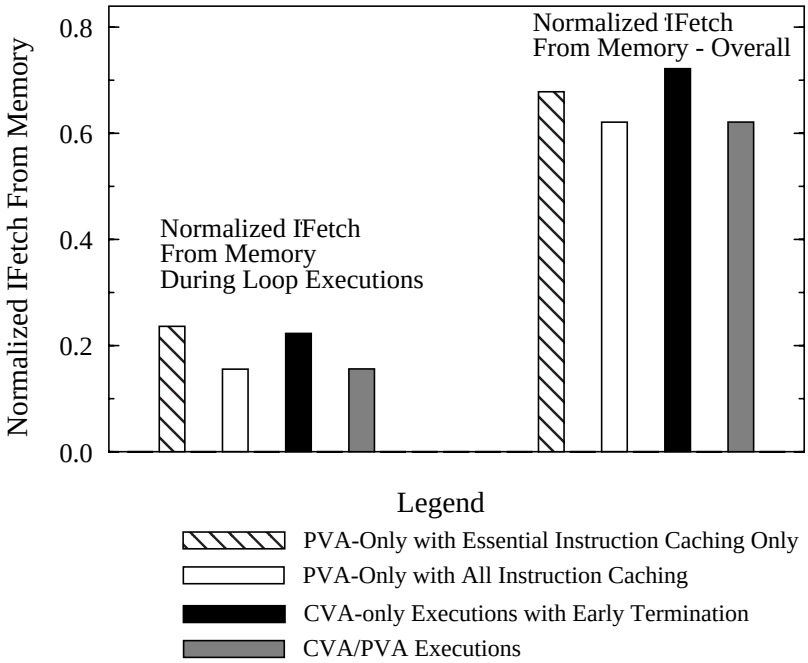


Figure 7.11: Normalized IFetch From Memory M0

Essential Instruction Only Caching vs. All Instruction Caching

For PVA executions, during loop executions, by also caching the non-essential instructions, the normalized IFetch was reduced from 0.2362 to 0.1556. For overall executions, the normalized IFetch was only reduced from 0.6783 to 0.6210

The difference in normalized IFetch between these two approaches was *not* as significant as what we had expected. This is due to a combination of the following two reasons. First, in all of these benchmarks, *after* a loop is being vectorized by a PVA construct, the loop body is typically *small*; that is, PVA vectorization was able to significantly reduce the loop size. Second, inside the PVA loop bodies, the cs-load and cs-store operations are so *frequent* that many of the instructions within the vectorized loops were essential instructions - they were always being cached, regardless of which loop caching scheme was used. As a results of these two factors, most of the instructions in the PVA loops were captured in the loop cache in both approaches.

PVA Executions vs. CVA Exactions

The abovementioned two observations (small PVA vectorized loops and almost all instructions in the PVA loops are essential) are also responsible for having little different, in normalized IFetch, between the PVA-only, CVA-only and CVA/PVA executions.

PVA-Only with All Instruction Caching vs. CVA/PVA Executions

For PVA-only executions with “all instruction caching”, the loop cache was able to absorb almost *all* the instruction requests from the processor core during loop executions. As a result, it achieved the same overall normalized IFetch with those for the CVA/PVA executions (0.6210 for both of them). Although they consume about the same access power at M0, the PVA executions, however, consume power accessing the loop cache while the CVA executions do not.

The normalized IFetch from the memory for various execution modes are summarized in Table 7.8. For PVA executions with “all instruction caching”, the loop cache was able to absorb most of the instruction requests from the core during loop executions. As a result, it achieved the same normalized IFetch with the CVA/PVA executions.

Table 7.8: Normalized IFetch From Memory M0

Execution Modes	During Loop Executions	Overall
PVA-Only with Essential Instruction Only Caching	0.2362	0.6783
PVA-Only with All Instruction Caching	0.1556	0.6210
CVA-Only (with Early Terminations)	0.2230	0.7218
CVA/PVA	0.1560	0.6210

7.6 Summary

The PVA-only executions improves the performance significantly by eliminating various types of loop execution overheads. The overall speedup, in this case, is 1.488. The CVA-only executions improves the performance even more significantly, with an overall speedup ranges between 1.585 and 1.617, depending on whether we allow a CVA vector execution to terminate early.

The CVA/PVA executions achieved the highest overall speedup (1.734). This was achieved on three fronts:

- For program loops that are highly vectorizable, we use the CVA construct to extract the maximum possible parallelism from the loops;
- For program loops that are impossible or costly to vectorize (in a conventional sense), we use the PVA construct to eliminate various types of overheads for loop executions;

- There are yet some loops that can be best vectorized using a combination of CVA and PVA constructs. An example of such loops was shown in Example 6.5 on page 102.

Vectorizations (using PVA or CVA constructs) reduced the instruction fetch bandwidth drastically. The CVA executions are inherently low in instruction fetch bandwidth. For PVA executions with the “all instruction caching” scheme, the normalized IFetch is about the same as those for CVA/PVA executions.

CHAPTER 8

ARCHITECTURAL EXTENSIONS FOR DSP APPLICATIONS

In this Chapter, we will describe some architectural extensions to the pseudo-vector machine for DSP applications. In particular, we will look at how these extensions can enhance the Infinite Impulse Response (IIR) Filter and Fast Fourier Transform (FFT) computations. Evaluating the performance benefits of these extensions, however, is beyond the scope of this dissertation.

8.1 Architectural Extensions - VLIW/Vector Machine

The pseudo-vector machine proposed in this dissertation can be extended by adding some processing capabilities in the “width” or “horizontal” direction, as shown in Figure 8.1.

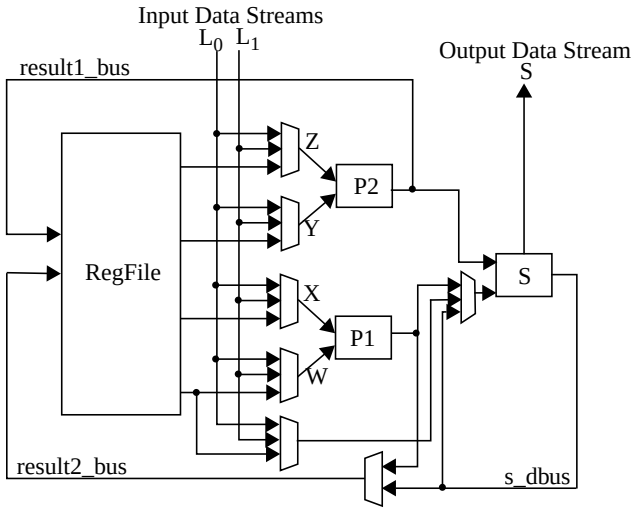


Figure 8.1: Datapath for the Extended Pseudo-Vector Machine

In this enhanced machine, there are two primary arithmetic functions, p1_op and p2_op, performed by two functional units, P1 and P2, respectively. With these two functional units, the machine can issue two independent operations to P1 and P2 simultaneously, and the two results

produced by them can be written back to the register file via two independent result buses (similar to a 2-wide VLIW machine).

The S unit is similar to those in the original pseudo-vector machine. It can perform some memory store operations (including the output data streaming operations) as well as some simple ALU arithmetic. However, the S unit now has three inputs (to perform three-input “add”, three-input “or”, etc.).

Like the original pseudo-vector machine, there are three types of CVA executions: compound CVA, reduction CVA and hybrid CVA. The dependency graphs for these three types of CVA are shown in Figure 8.2.

In these dependency graphs, operands W, X, Y and Z can *independently* source from (i) input stream L_0, or (ii) input stream L_1; or (iii) a designated register.

Each of these operands can also source from a zero-extended upper or lower halfword of an input stream L_0 or L_1, as defined in Table 7.9. This sourcing mode is depicted in this Table as “Cross Sourcing”.

Table 7.9: Possible Sources for Operands W, X, Y and Z

Operands	Operands Sourcing Modes	
	Independent Sourcing	Cross Sourcing
W	L_0, L_1, R0	{0,L_0[31:16]}
X	L_0, L_1, R4	{0,L_1[15:0]}
Y	L_0, L_1, R8	{0,L_1[31:16]}
Z	L_0, L_1, R12	{0,L_0[15:0]}

In this Table, {0, L_0[15:0]} denotes the zero-extended lower halfword from the input stream L_0. {0, L_0[31:16]} denotes the zero-extended upper halfword from the input stream L_0, etc.

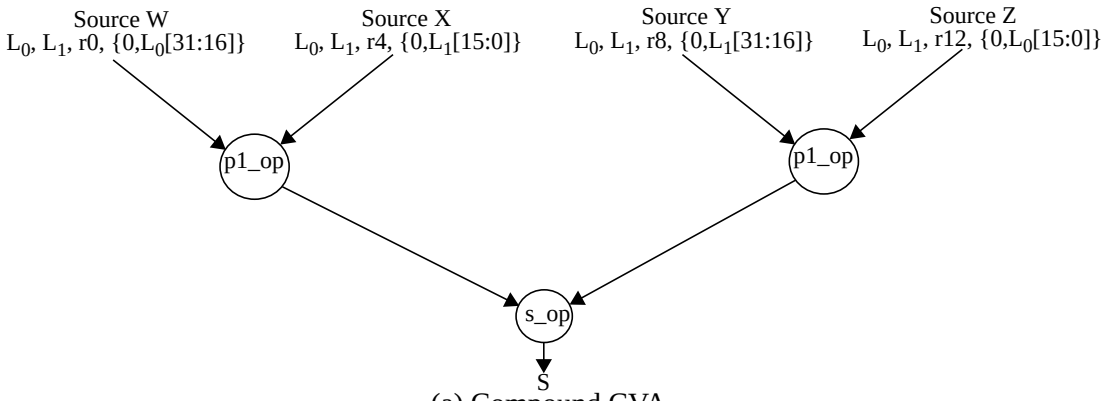
The three types of CVA depicted in Figure 8.2 have the following general forms.

- Compound CVA:

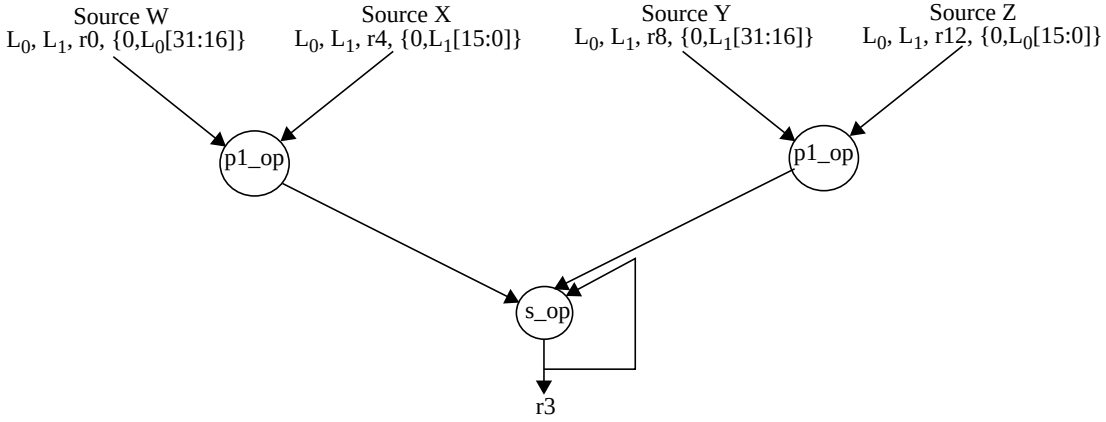
  $S_i = (W_i \text{ p1\_op } X_i) \text{ s\_op } (Y_i \text{ p2\_op } Z_i)$ i=0,...,n-1;

- Reduction CVA:

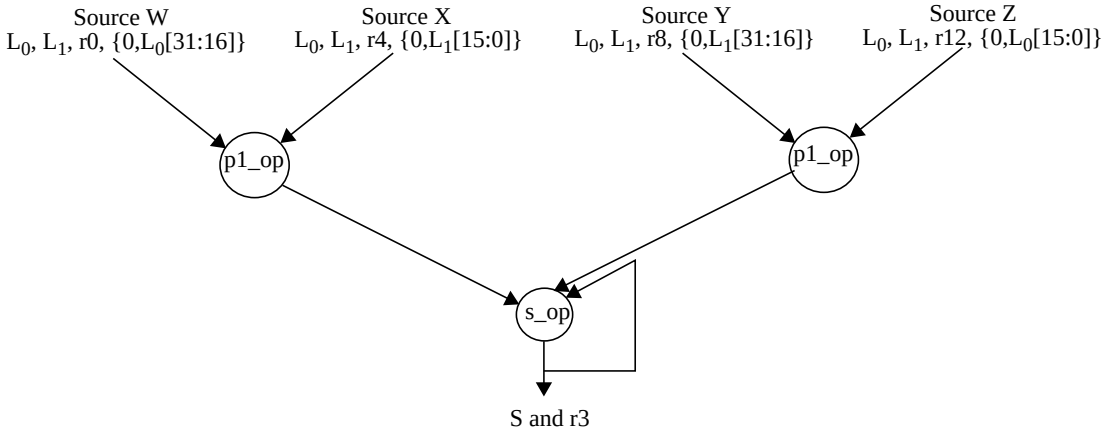
  $S_0 = (W_0 \text{ p1\_op } X_0) \text{ s\_op } (Y_0 \text{ p2\_op } Z_0)$;
  $S_i = (W_i \text{ p1\_op } X_i) \text{ s\_op } (Y_i \text{ p2\_op } Z_i) \text{ s\_op } S_{i-1}$, i=1,...,n-1;



(a) Compound CVA
(S enabled)



(a) Reduction CVA
(S disabled)



(a) Hybrid CVA
(S enabled)

Figure 8.2: Dependency Graphs For Three Types of CVA Executions

$R = S_{n-1}$

where

S_i denotes the *i*th partial result;

R denotes the scalar result for the vector reduction operations.

- Hybrid CVA:

Same as reduction CVA, except that S_i ,*i*=0,...,n-1, is also written to the memory via the S stream.

When p2_op is a “pass Z” function, this machine degenerates into the original pseudo-vector machine.

8.2 Implementing The IIR Filter

An *n*th order biquad infinite impulse response (IIR) filter is represented by the following z-domain transfer function.

$$H(z) = Y(z)/X(z) = (B_0 + B_1z^{-1} + B_2\, z^{-2} + \ldots + B_nz^{-n}) / (1 + A_1z^{-1} + A_2\, z^{-2} + \ldots + A_nz^{-n})$$

The corresponding difference equation is given by

$$Y(t) = B_0 * X[t] + \sum_{i = 1, n} (B_i * X[t-i] - A_i * Y[t-i]) \qquad (8.1)$$

To implement this filter, the input vector <X[t-1],X[t-2],...,X[t-*n*]>, the output vector <Y[t-1],Y[t-2],...,Y[t-*n*]> and the two coefficient vectors $<B_1,...,B_n>$ and $<A_1,...,A_n>$ are organized in the memory system as follows.

First, coefficients A_i are negated, such that $A_i' = -A_i$, *i*=1,...,*n*. A new vector is formed: $<A_1',...,A_n'>$.

The size of all the elements in these four vectors are assumed to be halfword long (16 bits). The input vector <X[t-1],X[t-2],...,X[t-*n*]> and the output vector <Y[t-1],Y[t-2],...,Y[t-*n*]> are *paired* together, element-wise, to form a new vector. The *i*th element of this new vector is given by {X[t-*i*],Y[t-*i*]}. Likewise, the coefficient vectors $<A_1',...,A_n'>$ and $<B_1,...,B_n>$ are paired together, element-wise, to form another new vector. The *i*th element of this new vector is given by $\{A_i',B_i\}$.

Due to these element pairing, the element size of theses two new vectors are a word long (32 bits), or twice the element size of the original vectors.

Figure 8.3 shows how the enhanced datapath can perform the *n*th-order IIR shown in equa-

tion (8.1).

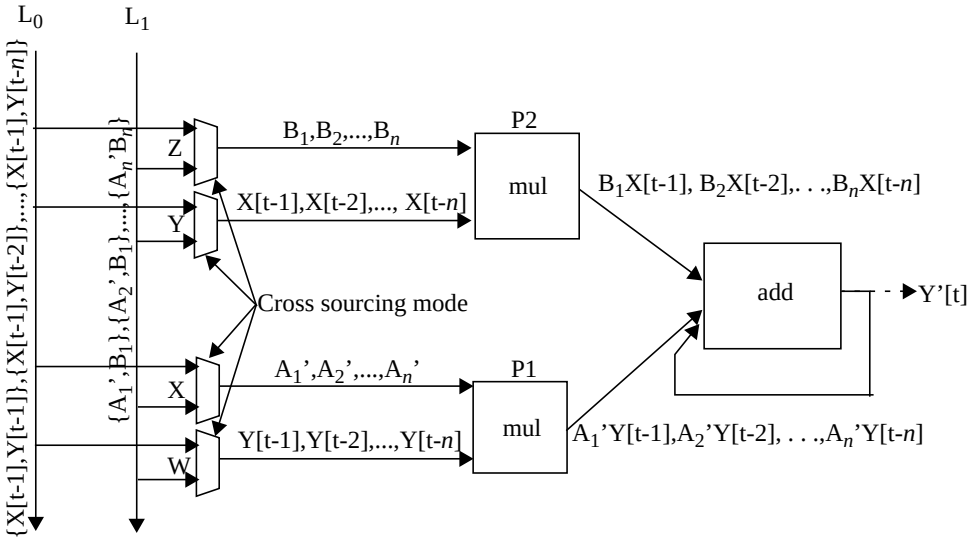


Figure 8.3: Enhanced Datapath For Implementing IIR Filters

In this figure, L_0 streams in <{X[t-1],Y[t-1]},{X[t-1],Y[t-2]},...,{X[t-1],Y[t-*n*]}>, while L_1 streams in <{A_1',B_1},{A_2',B_1},...,{A_n'B_n}>. The operands W, X, Y and Z, all in "cross-sourcing" mode, source in <A_1',A_2',...,A_n'>, <Y[t-1],Y[t-2],...,Y[t-*n*]>, <B_1,B_2,...,B_n> and <X[t-1],X[t-2],..., X[t-*n*]>, respectively.

The final reduction result is given by, Y'[t] = $\sum_{i = 1, n}$ (B_i * X[t-*i*] - A_i * Y[t-*i*]). A final term, B_0 * X[t], needs to be added to obtain the final result, Y[t].

The throughput rate of this implementation is *n*+C cycles per output sample, where C is some fixed vector startup cost.

The fetch addresses generated by L_0 and L_1 can also use *modulo addressing* to access circular buffers, identical to those used in [SHARC97,TMS320C3x]. These circular buffers are used to store sample stream <X[t-1],X[t-2],...,X[t-*n*]> and response stream <Y[t-1],Y[t-2],...,Y[t-*n*]>. These buffers can also used to store the coefficients A_i' and B_i, *i*=1,...,*n*.

8.3 Implementing The FFT

A N-point Discrete Fourier Transform (DFT) is defined as

$$H(k) = \sum_{n = 0}^{N - 1} h(n) e^{-j\frac{2\pi nk}{N}} \qquad k = 0,...,N-1 \qquad (8.2)$$

In this equation, N is assumed to be a power of 2; the time response, h(n), and the spectrum, H(k), are all complex numbers. In a radix two Decimation In Time (DIT) Fast Fourier Transform (FFT), a N-point DFT is broken down into two $\frac{N}{2}$-point DFTs. A $\frac{N}{2}$-point DFT is further broken down into two $\frac{N}{4}$-point DFTs, and so on. Equation(8.2) can be rewritten as:

The first half FFT,

$$H(k) = \left[\sum_{n=0}^{\frac{N}{2}-1} h_{2n} W_{\frac{N}{2}}^{nk}\right] + W_N^k \left[\sum_{n=0}^{\frac{N}{2}-1} h_{2n+1} W_{\frac{N}{2}}^{nk}\right] = H_0^1(k) + W^k H_1^1(k) \qquad k = 0,1,...,\frac{N}{2}-1;$$

and the second half FFT,

$$H\left(k+\frac{N}{2}\right) = \left[\sum_{n=0}^{\frac{N}{2}-1} h_{2n} W_{\frac{N}{2}}^{nk}\right] - W_N^k \left[\sum_{n=0}^{\frac{N}{2}-1} h_{2n+1} W_{\frac{N}{2}}^{nk}\right] = H_0^1(k) - W^k H_1^1(k) \qquad k = 0,1,...,\frac{N}{2}-1;$$

where $W_N = e^{-j\frac{2\pi}{N}}$; $H_0^1(k) = \sum_{n=0}^{\frac{N}{2}-1} h_{2n} W_{\frac{N}{2}}^{nk}$ and $H_1^1(k) = \sum_{n=0}^{\frac{N}{2}-1} h_{2n+1} W_{\frac{N}{2}}^{nk}$. The W_N^k term above is called the phase factor. The two $\frac{N}{2}$-point DFTs, $H_0^1(k)$ and $H_1^1(k)$, are combined in a butterfly, using the phase factors W^k and $-W^k$. This decomposition and recombination of a N-point DFT are illustrated in Figure 8.4.

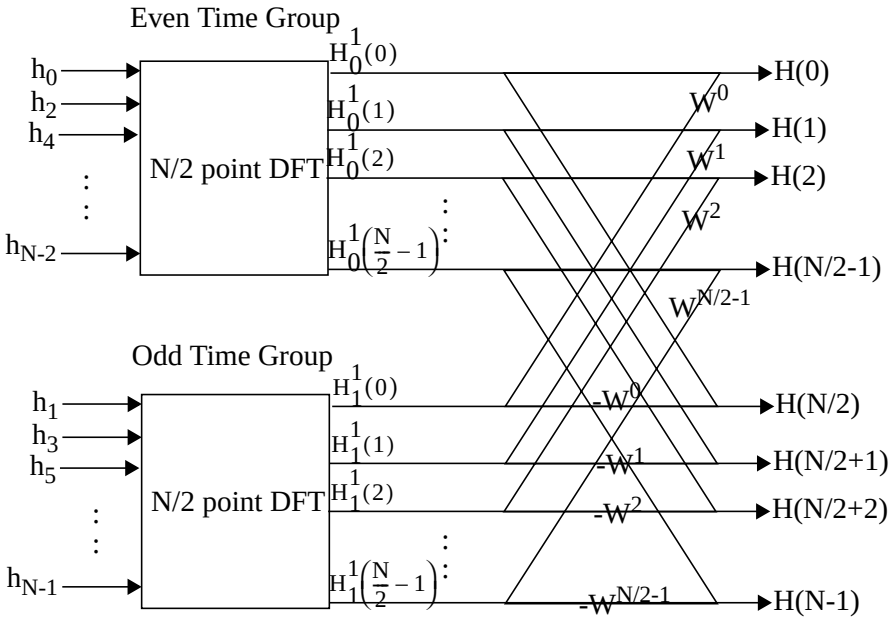


Figure 8.4: DIT Decomposition of a N-point FFT

Figure 8.5 shows the generalized butterfly computational structure for DIT FFT. In this figure, “Re” denotes the real part of a complex number; while “Im” denotes the imaginary part of the complex number.

Two complex data points, X(n) and X(m), are extracted from the memory. X(m) is multiplied by a complex exponential phase factor. The resulting real and imaginary parts are separated, combined with the respective parts of X(n) and written back to the memory . After the transformed data pair is written back to the memory, the process begins again on a different pair of memory locations, using an updated phase factor value. The process continues until the required number of butterflies has been computed.

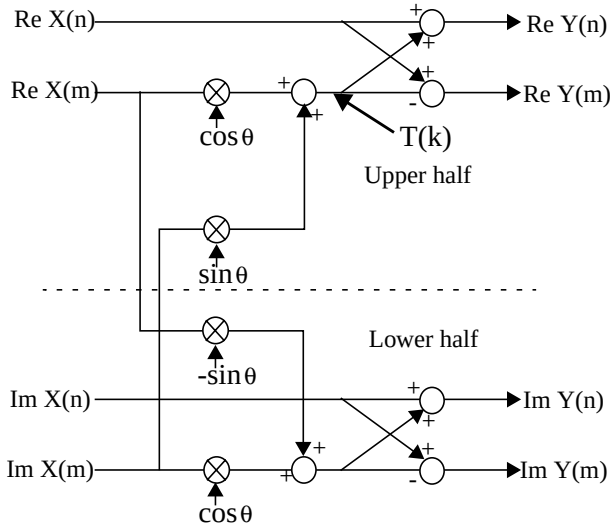


Figure 8.5: Generalized Butterfly Computation Diagram

The above computational structure can be implemented on our enhanced pseudo-vector machine as follows.

In this implementation, a complex number, X(n) = Re X(n) + j Im X(n), is represented using one word of data (32 bits): upper halfword for Re X(n); and lower halfword for Im X(n).

Two new vectors are created. Each element in these new vectors consists two phase factors. Each element in the first vector consists of a concatenation of two phase factors {$\cos\theta$, $\sin\theta$}; and each element in the second vector consists of a concatenation of two phase factors {$-\sin\theta$, $\cos\theta$}, where $\theta = \frac{2\pi i}{N}$, i=0,...,$\frac{N}{2}$-1. Each phase factor is assumed to be represented by a halfword data (16 bit); and each element in the new vectors is assumed to be 32 bit wide.

The computational structure shown in Figure 8.5 can be broken down into two halves: an upper half and a lower half, each consists of two multiplications and three additions. The two multiplications and the first addition can be mapped onto a compound CVA, as follows. To perform, say, the upper half of the butterfly, L_0 streams in <{Re X(0),Im X(0)},{Re X(1),Im X(1)},...,{Re X(N-1), Im X($\frac{N}{2}$-1)}>; while L_1 streams in <{Re W^0, Im W^0},{Re W^1,Im W^1},...,{Re $W^{N/2-1}$, Im $W^{N/2-1}$}>, or equivalently, <{cos 0, sin 0}, {cos$\frac{2\pi}{N}$, sin$\frac{2\pi}{N}$ }, {cos$\frac{4\pi}{N}$, sin$\frac{4\pi}{N}$ }, ... , {cos $\frac{2\pi\left(\frac{N}{2}-1\right)}{N}$, sin $\frac{2\pi\left(\frac{N}{2}-1\right)}{N}$ }>.

At the same time, operands W, X, Y and Z are all put in a “cross sourcing” mode, as shown in Figure 8.6.

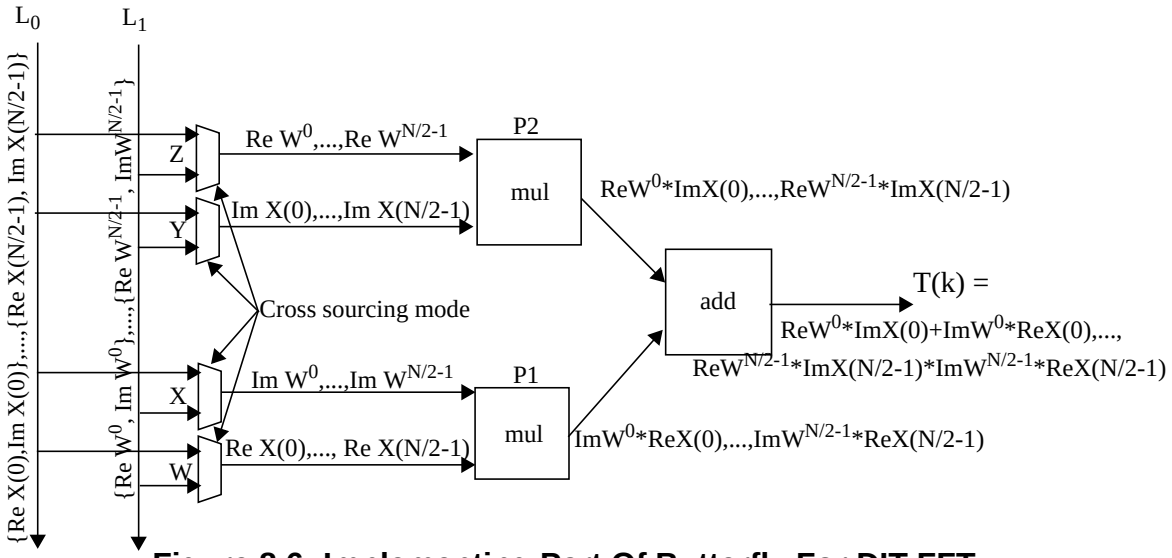


Figure 8.6: Implementing Part Of Butterfly For DIT FFT

The output of this compound CVA is a temporary vector, T(k), k=0,...,$\frac{N}{2}$-1. This vector is also shown in Figure 8.5. This vector is then added to the vector for Re X(n) to obtain the vector for Re Y(n); and subtract from the vector for Re X(n) to obtain the vector for Re Y(m). Three CVA instructions (one as shown in Figure 8.6, followed by two vector additions) are required for one half of the butterfly. To merge two $\frac{N}{2}$-point DFTs, six CVA instructions are required, or 6*N+C cycles, for some fixed constant cost C.

It should be noted that the vector computations (or the CVA executions) are only effective if the number of sample points are large (during the final stages of recombinations). When the decomposition is down to such a point where the number of sample points is small (2, 4 or 8, etc.), then a conventional loop-based DSP style of executions will be more efficient. This can be carried out on this machine using the PVA executions.

CHAPTER 9

SUMMARY

Many today's mobile applications require the underlaying execution machines to take advantage of the parallelism that frequently found in these applications. But in some other point in time, they also require the machines to perform control intensive functions. Besides performance, the design of these machines is also severely constrained by the *hardware costs* and *power consumption*.

In this dissertation, we proposed a processing paradigm, called the *pseudo-vector machine*, for executing these applications. This machine attempts to exploit the *low-power* and *high performance* aspects of vector processing paradigm to efficiently extract and exploit the ILP in the applications, *whenever possible*. The strength of vector processing arises from:

- The ability to pipeline various operations on a data stream (to improve performance);
- Result produced by a functional unit is routed directly to its destination functional unit, instead of being broadcast to the entire datapath;
- Lower instruction fetch bandwidth.

Efficient data storage and movements and lower instruction fetch bandwidth could also mean lower power consumption. The strength of vector processing makes it very suitable for the low-cost, low-power embedded computing systems.

Various design aspects and some of their unique features of this machine are summarized below [Lee99e,Lee99f].

Sharing of Functional Units Between Scalar and Vector Executions

In this machine, both scalar and vector executions are performed using a single, integrated datapath. In particular, both scalar and vector execution modes use the same set of functional units.

Thus, arithmetic functions that are available to scalar executions, are also available to vector executions - an efficient use of hardware resources.

CVA vs. PVA Executions - Choosing The Best Execution Mode For A Given Critical Loop

The vector execution mode, in this machine, can be further divided into *Canonical Vector Arithmetic* (CVA) mode and *Pseudo-Vector Arithmetic* (PVA) mode. The former is used for a "true" vector processing paradigm; while the latter is used for program loops that are difficult or impossible to vectorize (in a conventional sense).

Two vector instructions are added to the M-CORE ISA: a CVA instruction and a PVA instruction. These instructions can optionally and independently, enable or disable two input data streams (L_0 and L_1) and one output data stream (S).

There three types of CVA executions on this machine: *compound CVA*; *reduction CVA* and *hybrid CVA*. These three types of CVA represent some basic vector arithmetic commonly found in DSP and scientific computations.

For PVA executions, the heads of the input data queues are accessed by reading from register R0 and R1. In addition, an instruction in the PVA loop body can be selected as the instruction that will enqueue data to the output stream - the result written back by this instruction is used as the data for the output stream. This instruction can be specified by using a label called "cs-store" located within the loop body; or, equivalently, by specifying the index of the instruction (relative to the PVA instruction) in the PVA instruction itself.

Table 9.1 summarizes the speedups for various execution modes on this machine.

Table 9.1: Speedups For Various Execution Modes

Execution Modes		Without a TM		With a 512-byte TM	
		Overall Speedups	Speedups During Loop Executions	Overall Speedups	Speedups During Loop Executions
PVA-Only Executions[1]		1.488	3.692	1.488	3.692
CVA-Only Executions	Without early termination	1.585	4.002	N.A.[2]	N. A.[2]
	With early termination	1.560	4.340	1.617	4.617
CVA/PVA Executions		1.673	4.306	1.734	4.396

1. Not affected by TM.
2. Results are not available.

When executing in a PVA mode, the performance can be significantly improved by removing the overheads for: (i) loop control; (ii) cs-load operations; and (ii) cs-store operations. The overall speedup, by using *only* the PVA construct, is 1.488.

By using *only* the CVA construct, the overall speedup improves the performance even more significantly. The overall speedup for CVA-only executions with early termination is 1.617. The CVA executions allow the executions of multi-cycle functions from different iterations (in the original loop) to overlap in time.

When combining both CVA and PVA executions, the performance improves even further. The overall speedup, in this case, is 1.734. For loops that are highly vectorizable, it is best to vectorize them using the CVA construct. For loops that are not CVA vectorizable, PVA construct can be used instead. In addition, there are also certain loops that are best vectorized using a combination of both constructs. By allowing the use of both vectorizing constructs, a compiler is given all the flexibilities to vectorize a program loop, to achieve the best possible speedup and power consumption.

Using The Temporary Memory For Better Performance and Lower Power

Performance of vector executions is frequently limited by the memory bandwidth. Increasing the bandwidth by adding the number of memory read and write ports is an expensive proposition. In this dissertation, we proposed using a small Temporary Memory (TM) to increase the effective memory bandwidth during CVA executions. With the use of a 512-byte TM, the CVA executions can perform two data reads and one data write, all in a single cycle.

In some sense, TM has replaced the roles of vector registers in the conventional vector machines. Like its vector register counterpart, TM can be used to improve performance (by reducing the memory conflicts), as well as to save access power to the larger memory modules.

Compared with the vector registers, however, TM is more flexible, in terms of allocating and organizing these temporary storage spaces. Vector registers in the conventional vector machines typically have a fixed size and length. With the TM, a compiler can trade off between the vector length and the number of vectors that can be allocated into TM. A drawback of using TM, however, is the extra overheads in specifying the vector length, strides and element sizes prior to each vector execution. These overheads, however, can be easily amortized over the executions of longer vectors.

Without TM, performance for vector operations that perform two data reads and one data write per cycle will be impacted. Their throughput rates, in this case, were degraded from one result per cycle, to one result every two cycles. For CVA/PVA executions, the removal of TM reduces the overall speedup from 1.734 to 1.673.

Register Overlay and Temporary Registers

In this dissertation, we had also discussed various aspects of machine states maintenance, with respect to interrupt on vector executions. We introduced the concept of *register overlay* and *temporary registers*, in an attempt to reduce the interrupt response latency for vector executions. The latter is an important design factor for many real-time applications.

A temporary register used to store temporary data streamed in from the memory; while an overlaid register is used to store the load addresses corresponding to the data fetched into its temporary register. During a context switch, all temporary registers are not saved as part of the vector execution contexts.

Instruction Fetch Memory Bandwidth Reductions

Vectorization drastically reduces the instruction fetch bandwidth. The CVA execution, in particular, is inherently low in this bandwidth requirement. After fetching all the setup and initialization code, a CVA execution has no further instruction request for the rest of its vector executions.

Vectorization via a PVA construct also reduces the instruction fetch bandwidth drastically. It does so by reducing the loop size significantly. This involves removing from the loop body, some instructions that specify certain repetitive operations, such as cs-load, cs-store, and loop control operations.

The instruction fetch bandwidth for PVA executions can be further reduced by using a loop cache that attempts to cache *all* the instructions, not just those instructions that will cause a memory conflict with data references. This loop caching scheme, proposed in this dissertation, gives first priority to those instructions that will cause a data reference conflict. After these essential instructions are allocated into the loop cache, the latter then tries to capture all other instructions, without replacing any essential instructions.

Using this loop caching scheme, the PVA executions can absorb *almost all* the instruction requests coming from the processor core during loop executions.

Appendix A: Critical Loop Vectorizations and Cycle Saving Calculations

A.1 Benchmark “auto”

A.1.1 Critical Loop 1

Table A.1.1: Vectorizing Critical Loop 1

Address Opcode//; Assembly Code	Using PVA Construct	Using CVA Construct

00000308 940b //; stw r4,(r11)
0000030a 203b //; addi r11,4
0000030c 2004 //; addi r4,1
0000030e 0dd4 //; cmplt r4,r13
00000310 e7fb //; bt 0x0000308
	PVA @S, #1
	cs-store:
	addi r4,1[a]
		movi r4,1
		CVA mov r4, @P,
		add @P, r3, {r3,@S}
		. . . . . .
		movi r5, 1
		CVA add @L0, r5, @S

a. Essential instruction

Table A.1.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000308	target	2100	-	-	-
00000310	bt	2100	00000308	2097 (99.9)	3 (0.143)

This loop performs vector initialization, C[i] = i, where i=0,1,2,3...
Estimated execution cycles = 7 x 2097 + 6 x 3 = 14697
Average number of iterations per invocation = 2100/3 = 700

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup/exit costs = 8 x 3 + 1(M0 conflict) = 25
cs-store saving: 3 x 2100 - 25/2 = 6288
lp-ctl saving: 3 x 2097 + 2 x 3 - 25/2 = 6285
Total saving = 6288 + 6285 = 12573

(ii) CVA-Only executions

A temporary vector is created. This temporary vector can be stored in M0 or M1. TM is not used.
First CVA instruction (t_p=t_s=1):
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit)
+ 1(t_p) + 1(t_s) - 1 + 1(extra “mov” inst.)= 10
Execution time: 3 x 10 + 2100 = 2130
Second CVA instruction (t_p=1, t_s=0):
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2)
+ 1(exit) + 1(t_p) + 0(t_s) - 1 + 1(extra “mov” inst.)= 10
Execution time: 3 x 10 + 2100 = 2130
CVA-only saving = 14697 - 2130 - 2130 = 10437

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machine:
IReq from core during loop executions: 6 x 2097 + 5 x 3 = 12597
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 3 + 1 x 2100 = 2118
CVA-only executions:
IReq from core during CVA executions: 7 x 2 (setup code/vector inst.) x 3 = 42

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 3 + 1 (essential inst.) = 19
PVA-only with all inst. caching: same as PVA-only with essential inst. caching.
CVA/PVA executions: 7 x 2 (setup code/vector inst.) x 3 = 42

A.1.2 Critical Loop 2:

Table A.1.3: Vectorizing Critical Loop 2

Address Opcode//; Assembly Code	Using PVA Construct	Using CVA Construct
000010d2 950e //; stw r5,(r14) 000010d4 203e //; addi r14,4 000010d6 01b6 //; decne r6 000010d8 e7fc //; bt 0x010d2	PVA @S, #1 cs-store: mov r5, r5[a]	CVA mov r5, @S;

a. Essential instruction

Table A.1.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
0000010d2 0000010d8	target bt	705 705	- 000010d2	- 704 (99.9)	- 1 (0.142)

This loop performs vector initialization, C[i] = r5, for some scalar r5.
Estimated execution cycles = 6 x 704 + 5 x 1 = 4229
Average number of iterations per invocation = 705/1 = 705

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup/exit costs = 8 x 1 + 1(M0 conflict) = 9
cs-store saving: 2 x 705 - 9/2 = 1406
lp-ctl Saving: 3 x 704 + 2 x 1 - 9/2 = 2110
Total saving: 1406 + 2110 = 3516

(ii) CVA-Only executions
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2)

$+ 1(\text{exit}) + 1(t_p) + 0(t_s) - 1 = 8$

Execution time = 8 x 1 + 705 = 713
CVA saving = 4229 - 713 = 3516

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machine:
IReq from core during loop executions: 6 x 704 + 5 x 1 = 4229
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 1 + 1 x 705 = 711
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 1 = 6

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 1 + 1 (essential inst.) = 7
PVA-only with all inst. caching: same as PVA-only with essential inst. caching.
CVA/PVA executions: 6 (setup code/vector inst.) x 1 = 6

A.1.3 Summary

Total program execution cycles: 26381
Total number of cycles in loops: 14697 (loop 1) + 4229 (loop 2)
= 18926 or 71.76% of program execution time.
Average number of iterations per invocation = (700+705)/2 = 702.5

(i) PVA-Only executions
cs-store saving: 6288 + 1406 = 7694
lp-ctl Saving: 6285 + 2110 = 8395
Total cycle saving: 7694 + 8395 = 16089;%cycle saving: 16089/26381=0.6099
Speedup during loop executions = 18926/(18926 - 16089) = 6.671 (perf. imp. = 0.8501)
Overall speedup: 26381/(26381 - 16089) = 2.563 (perf. imp. = 0.6098)

Table A.1.5: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	8395	0.3182
cs-load-oh	0	0
cs-store-oh	7694	0.2917
Total	16091	0.6099

(ii) CVA-Only executions
Total cycle saving: 10437 + 3516 = 13953
Speedup during loop executions = 18926/(18926-13953)= 3.806 (perf. imp. = 0.7373)
Overall speedup = 26381/(26381 - 13953) = 2.123 (perf. imp. = 0.5290)

(iii) CVA/PVA executions

Total cycle saving: 12573(PVA for loop1) + 3516(CVA for loop2) = 16089
Speedup during loop executions = 18926/(18926 - 16089) = 6.671(perf. imp. = 0.7373)
Overall speedup: 26381/(26381 - 16089) = 2.563 (perf. imp. = 0.6098)

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machine:
IReq from core overall: 20695
IReq from core during loop executions: 12597 + 4229 = 16826

PVA-only executions:
IReq from core reduced = 16826 - (2118 + 711) = 13997
Normalized IReq from core during PVA executions = (2118 + 711)/16826 = 0.168
IReq from core overall = 20695 - 13997 = 6698
Normalized IReq from core overall = (20695 - 13997)/20695 = 0.3237

CVA-only executions:
IReq from core reduced = 16826 - (42 + 6) = 16778
Normalized IReq from core during CVA executions = (42+6)/16826 = 2.853×10^{-3}
Normalized IReq from core overall = (20695 - 16778)/20695 = 0.1893

CVA/PVA executions:
IReq from core reduced = 16826 - (42 + 6) = 16778
Normalized IReq from core during CVA/PVA executions = 48/16826 = 2.853×10^{-3}
Normalized IReq from core overall = (20695 - 16778)/20695 = 0.1893

Table A.1.6: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.168	2.853×10^{-3}	2.853×10^{-3}	0.3237	0.1893	0.1893

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory reduced = 16826 - (19 + 7) = 16800
Normalized Ifetch from memory during PVA executions: (19 + 7)/16826 = 1.545×10^{-3}
Ifetch from memory overall = 20695 - 16800 = 3895
Normalized Ifetch from memory overall = (20695 - 16800)/20695 = 0.1882

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:
Ifetch from memory: same as those for IReq from core
CVA/PVA executions:
Ifetch from memory: same as those for IReq from core

Table A.1.7: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
$1.545x10^{-3}$	$1.545x10^{-3}$	2.853×10^{-3}	2.853×10^{-3}	0.1882	0.1882	0.1893	0.1893

A.2 Benchmark “blit”

A.2.1 Critical Loop 1

Table A.2.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

00000304 8a0e //; ldw r10,(r14)
00000306 25f4 //; decne r4
00000308 12a7 //; mov r7,r10
0000030a 0b97 //; lsr r7,r9
0000030c 1e37 //; or r7,r3
0000030e 12a3 //; mov r3,r10
00000310 970d //; stw r7,(r13)
00000312 1b13 //; lsl r3,r1
00000314 203e //; addi r14,4
00000316 203d //; addi r13,4
00000318 e7f4 //; bt 0x0000304
	PVA @L0,@S, #5
	mov r3, r0
	mov r7, r3
	lsr r7,r9
	lsl r3,r1
	cs-store:
	or r3,r7[a]
		Setup a temporary vector T
		. . . . . .
		; T[i] = lsr(A[i], r9)
		mov r5, r9
		CVA lsr @L0, r5, @S;
		. . . . . .
		;@L1 <---- T[i]
		mov r5, r1
		CVA lsl @L0, r5, @P,
		or @L1, @P, @S;

[a] Essential instructions

Table A.2.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000304 00000318	target bt	3000 3000	- 00000304	- 2997 (99.9)	- 3 (0.100)

This loop performs vector operations C[i] = lsr(A[i], r9) | lsl(A[i], r1), for some scalars r1 and r9.
Estimated execution cycles = 14 x 2997 + 13 x 3 = 41997
Average number of iterations per invocation = 3000/3 = 1000

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) = 9 cycles
Total setup/exit costs = 9 x 3 + 1(M0 conflict) = 28
Cycles per iteration: 5
Cycle saving per iteration = 14 - 5 = 9 (3 for lp-ctl, 3 for cs-load, 3 for cs-store)
cs-load saving: 3 x 3000 - 28/3 = 8991
cs-store saving: 3 x 3000 - 28/3 = 8991
lp-ctl saving: 3 x 2997 + 2 x 3 - 28/3 = 8988
Total saving = 8991 + 8991 + 8988 = 26970

(ii) CVA-Only executions

A temporary vector is stored in TM. Strip-mining for TM is needed (see Section 6.8 on page 104).
First CVA instruction (t_p=1, t_s=0):
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 1(t_p) + 0(t_s) - 1 + 1(extra “move” inst.) = 10
Second CVA instruction (t_p=1, t_s=1):
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) + 1(t_p) + 1(t_s) - 1 + 1(extra “move” inst.) = 12 cycles
Setup/exit costs per invocation (C_{CVA}) = 10 + 12 = 22 cycles per invocation
In each invocation, vector length n=1000.

Execution time per invocation = 24 + (n/128+1) x (43 + C_{CVA}) + 2n = 2544
There are a total of 3 invocations.
CVA-only saving = 41997 - 3 x 2544 =

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
TM is used to store the temporary vector.
C_{CVA}=22; n=1000;
With no TM:
Execution time: C_{CVA} + n + 2n = 3022
With TM = 256 bytes:
Execution time using strip-mining: 24 + (n/64+1) x (43 + C_{CVA}) + 2n = 3064 (> execution time without using TM)
With TM = 1024 bytes: no stripe mining necessary.
Execution time: C_{CVA} + n + n = 2022

Table A.2.3: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	Using TM?	Exe. Time per Invocation	Total Exe. Time	CVA-only Cycle Saving
0	N	3022	9066	32931
64	N	3022	9066	32931
128	N	3022	9066	32931
256	N	3022	9066	32931
512	Y	2544	7632	34365
1024	Y	2022	6066	35931

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 12 x 2997 + 11 x 3 = 35997
PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 3 + 5 x 3000 = 15021
CVA-only executions:
IReq from core during CVA executions: 15 (setup code/vector inst.) x 3 = 45

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 3 + 1 (essential inst.)
+ 4 x 3000 (non-essential inst.) = 12021
PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 3 + 1 (essential inst.)
+ 4 x 4 (non-essential inst.) = 38
CVA-only executions: 15 (setup code/vector inst.) x 3 = 45
CVA/PVA executions: 15 (setup code/vector inst.) x 3 = 45

A.2.2 Critical Loop 2

Table A.2.4: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

00000356 1227 //; mov r7,r2
00000358 0b17 //; lsr r7,r1
0000035a 1e37 //; or r7,r3
0000035c 970d //; stw r7,(r13)
0000036e 1223 //; mov r3,r2
00000360 820e //; ldw r2,(r14)
00000362 25f4 //; decne r4
00000364 1b93 //; lsl r3,r9
00000366 203e //; addi r14,4
00000368 203d //; addi r13,4
0000036a e7f4 //; bt 0x00000356
	PVA @L0,@S, #5
	mov r3, r0[a]
	mov r7, r3[a]
	lsr r7,r9
	lsl r3,r1
	cs-store:
	or r3,r7[a]
		Setup a temporary vector T
		. . . . . .
		; T[i] = lsr(A[i], r9)
		mov r5, r1
		CVA lsr @L0, r5, @S;
		. . . . . .
		;@L1 <---- T[i]
		mov r5, r9
		CVA lsl @L0, r5, @P,
		or @L1, @P, @S;

[a] Essential instructions

Table A.2.5: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000356 0000036a	target bt	3000 3000	- 00000356	- 2997 (99.9)	- 3 (0.100)

This loop performs vector operations C[i] = lsr(A[i], r1) | lsl(A[i], r9), for some scalars r1 and r9.
Estimated execution cycles = 14 x 2997 + 13 x 3 = 41997
Average number of iterations per invocation = 3000/3 = 1000

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 2
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) = 9 cycles
Total setup/exit costs = 9 x 3 + 2(M0 conflicts) = 29
Cycles per iteration: 6
Cycle saving per iteration = 14 - 5 = 9
cs-load saving: 3 x 3000 - 29/3 = 8990
cs-store saving: 3 x 3000 - 29/3 = 8990
lp-ctl saving: 3 x 2997 + 2 x 3 - 29/3 = 8987
Total saving = 8990 + 8990 + 8987 = 26967

(ii) CVA-Only executions

First CVA instruction (t_p=1, t_s=0):
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 1(t_p) + 0(t_s) - 1 + 1(extra "move" inst.) = 10 cycles
Execution time = 3 x 10 + 3000 = 3030
Second CVA instruction (t_p=1, t_s=1):
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) + 1(t_p) + 1(t_s) - 1 + 1(extra "move" inst.) = 12 cycles
Setup/exit costs per invocation (C_{CVA}) = 10 + 12 = 22 cycles per invocation
In each invocation, vector length n=1000.
Execution time per invocation = 24 + (n/128+1) x (43 + C_{CVA}) + 2n = 2544
There are a total of 3 invocations.

CVA-only saving = 41997 - 3 x 2544 = 34365

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
TM is used to store the temporary vector.
CCVA=22; n=1000;
With no TM:
Execution time: C_{CVA} + n + 2n = 3022
With TM = 256 bytes:
Execution time using strip-mining: 24 + (n/64+1) x (43 + C_{CVA}) + 2n = 3064 (> execution time without using TM)
With TM = 1024 bytes: no stripe mining necessary.
Execution time: C_{CVA} + n + n = 2022

Table A.2.6: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	Using TM?	Exe. Time per Invocation	Total Exe. Time	CVA-only Cycle Saving
0	N	3022	9066	32931
64	N	3022	9066	32931
128	N	3022	9066	32931
256	N	3022	9066	32931
512	Y	2544	7632	34365
1024	Y	2022	6066	35931

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 12 x 2997 + 11 x 3 = 35997
PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 3 + 5 x 3000 = 15021
CVA-only executions:
IReq from core during CVA executions: 15 (setup code/vector inst.) x 3 = 45

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 3 + 3 (essential inst.)
+ 4 x 3000 (non-essential inst.) = 12021

PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 3 + 3 (essential inst.)
+ 2 x 4 (non-essential inst.) = 32

CVA-only executions: 15 (setup code/vector inst.) x 3 = 45
CVA/PVA executions: 15 (setup code/vector inst.) x 3 = 45

A.2.3 Summary

Total program execution cycles: 84739
Total number of cycles in loops: 41997 (loop 1) + 41997 (loop 2)= 83994

or 99.18% of program execution time.
Average number of iterations per invocation = 1000

(i) PVA-Only executions
cs-store saving: 8991 + 8990 = 17981
cs-store saving: 8991 + 8990 = 17981
lp-ctl Saving: 8988 + 8987 = 17975
Total cycle saving: 17981 + 17981 + 17975 = 53937
Speedup during loop executions = 83994/ (83994 - 53937) = 2.794 (perf. imp. = 0.6421)
Overall speedup: 84739/(84739 - 53937) = 2.751 (perf. imp. = 0.6365)

Table A.2.7: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	17975	0.2121
cs-load-oh	17981	0.2122
cs-store-oh	17981	0.2122
Total	53937	0.6365

(ii) CVA-Only executions
Total cycle saving: 34365 (loop 1) + 34365 (loop 2) = 68730
Speedup during loop executions = 83994/(83994-68730)= 5.503 (perf. imp. = 0.8183)
Overall speedup = 84739/(84739 - 68730) = 5.293 (perf. imp. = 0.8111)

(iii) CVA/PVA executions
Total cycle saving: 34365 (CVA loop 1) + 34365 (CVA loop 2) = 68730
Speedup during loop executions = 83994/(83994-68730)= 5.503 (perf. imp. = 0.8183)
Overall speedup = 84739/(84739 - 68730) = 5.293 (perf. imp. = 0.8111)

(iv) CVA Executions Using Various Sizes of TM

Table A.2.8: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	CVA-Only Executions			CVA/PVA Executions		
	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup
0	65862	4.632	4.489	65862	4.632	4.489
64	65862	4.632	4.489	65862	4.632	4.489
128	65862	4.632	4.489	65862	4.632	4.489
256	65862	4.632	4.489	65862	4.632	4.489
512	68730	5.503	5.293	68730	5.503	5.293
1024	71862	6.923	6.581	71862	6.923	6.581
2048	71862	6.923	6.581	71862	6.923	6.581
4096	71862	6.923	6.581	71862	6.923	6.581

(v) IReq From Core

Base machinne:
IReq from core overall: 78448
IReq from core during loop executions: 35997 + 35997 = 71994

PVA-only executions:
IReq from core reduced = 71994 - (15021 + 15021) = 41952
Normalized IReq from core during PVA executions = (15021 + 15021)/71994 = 0.4173
Normalized IReq from core overall = (78448 - 41952)/78448 = 0.4652

CVA-only executions:
IReq from core reduced = 71994 - (45 + 45) = 71904
Normalized IReq from core during CVA executions = (45 + 45)/71994 = 1.250×10^{-3}
Normalized IReq from core overall = (78448 - 71904)/78448 = 0.0834

CVA/PVA executions:
IReq from core reduced = 71994 - (45 + 45) = 71904
Normalized IReq from core during CVA/PVA executions = (45 + 45)/71994 = 1.250×10^{-3}
Normalized IReq from core overall = (78448 - 71904)/78448 = 0.0834

Table A.2.9: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.4173	1.250×10^{-3}	1.250×10^{-3}	0.4652	0.0834	0.0834

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory reduced = 71994 - (12021 + 12021) = 47952
Normalized Ifetch from memory during PVA executions: (12021 + 12021)/71994 = 0.339
Normalized Ifetch from memory overall = (78448 - 47952)/78448 = 0.3887

PVA-only with all inst. caching:
Ifetch from memory reduced = 71994 - (38 + 32) = 71924
Normalized Ifetch from memory during PVA executions: (38 + 32)/71994 = 9.723×10^{-4}
Normalized Ifetch from memory overall = (78448 - 71924)/78448 = 0.0832

CVA-only executions:
Ifetch from memory: same as those for IReq from core

CVA/PVA executions:
Ifetch from memory: same as those for IReq from core

Table A.2.10: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.339	9.723×10^{-4}	1.250×10^{-3}	1.250×10^{-3}	0.3887	0.0832	0.0834	0.0834

A.3 Benchmark “compress”

A.3.1 Critical Loop 1

Table A.3.1: Vertorizing Critical Loop 1

Address	Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000c32 00000c34 00000c36 00000c38	950e //; stw r5,(r14) 203e //; addi r14,4 01b6 //; decne r6 e7fc //; bt 0x00000c32	PVA @S, #1 cs-store: mov r5, r5[a]	CVA mov r5, @S;

a. Essential instruction

Table A.3.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000c32 00000c38	target bt	7969 7969	- 00000304	- 7968 (100)	- 1 (0.00)

This loop performs vector initialization, C[i] = r5, for some scalar r5.
Estimated execution cycles = 6 x 7968 + 5 x 1 = 47813 or 9.81% execution time.
Average number of iterations per invocation = 7969

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8 cycles
Total setup/exit costs = 8 x 1 + 1(M0 conflict) = 9
Cycles per iteration: 6
Cycle saving per iteration = 5
cs-store saving: 2 x 7969 - 9/2 = 15934
lp-ctl saving: 3 x 7968 + 2 x 1 - 9/2 = 23902
Total saving = 15934 + 23902 = 39836
Speedup during loop executions = 47813 / (47813 - 39836) = 5.994 (perf. imp. = 0.8332)
Overall speedup = 487021/(487021-39836) = 1.089 (perf. imp. = 0.0817)

Table A.3.3: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	23902	0.0491
cs-load-oh	0	0
cs-store-oh	15934	0.0327
Total	39836	0.0818

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit)
+ 1(t_p) + 0(t_s) - 1 = 8
Execution time = 8 x 1 + 7969 = 7977
CVA saving = 47813 - 7977 = 39836

Speedup during loop executions = 47813 / (47813 - 39836) = 5.994 (perf. imp. = 0.8332)
Overall speedup 487021/(487021-39836) = 1.089 (perf. imp. = 0.0817)

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core overall: 355216
IReq from core during loop executions: 5 x 7968 + 4 x 1 = 39844

PVA-only executions:
IReq from core during PVA executions = 6 x 1 + 1 x 7969 = 7975
IReq from core reduced = 39844 - 7975 = 31869
Normalized IReq from core during PVA executions = 7975/39844 = 0.2002
Normalized IReq from core overall = (355216 - 31869)/355216 = 0.9103

CVA-only executions:
IReq from core during CVA executions = 6 x 1 = 6
IReq from core reduced = 39844 - 6 = 39838
Normalized IReq from core during CVA executions = 6/39844 = 1.506×10^{-4}
Normalized IReq from core overall = (355216 - 39838)/355216 = 0.8878

CVA/PVA executions:
IReq from core during CVA/PVA executions = 6 x 1 = 6
IReq from core reduced = 39844 - 6 = 39838
Normalized IReq from core during CVA/PVA executions = 6/39844 = 1.506×10^{-4}
Normalized IReq from core overall = (355216 - 39838)/355216 = 0.8878

Table A.3.4: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.2002	1.506×10^{-4}	1.506×10^{-4}	0.9103	0.8878	0.8878

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 1 + 1 (essential inst.) = 7
Ifetch from memory reduced = 39844 - 7 = 39837
Normalized Ifetch from memory during PVA executions: 7/39844 = 1.757×10^{-4}
Normalized Ifetch from memory overall = (355216 - 39837)/355216 = 0.8878

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:
IReq from core = Ifetch from memory

CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.3.5: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
1.757×10^{-4}	1.757×10^{-4}	1.506×10^{-4}	1.506×10^{-4}	0.8878	0.8878	0.8878	0.8878

A.4 Benchmark “des”

A.4.1 Critical Loop 1

Table A.4.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
000008ac a702 //; ldb r7,(r2) 000008ae a603 //; ldb r6,(r3) 000008b0 0f67 //; cmpne r7,r6 000008b2 01c2 //; clrt r2 000008b4 e005 //; bt 0x008c0 000008b6 2002 //; addi r2,1 000008b8 2003 //; addi r3,1 000008ba 0184 //; declt r4 000008bc eff7 //; bf 0x008ac	PVA @L0,@L1, ct=1,#1 cmpne r0, r1[a]	CVA cmpne.ct=1 @L0,@L1;

a. Essential instruction

Table A.4.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000008ac	target	1269	-	-	-
000008b4	bt	1269	000008b6	0 (0)	1269 (100)
000008ba	target	1269	-	-	-
000008bc	bt	1269	000008ac	1128 (88.9)	141 (11.1)

This loop reads two vectors sequentially and finds the first i that satisfies: A[i] != B[i].
Estimated execution cycles = 12 x 1128 + 11 x 141 = 15087
Average number of iterations per invocation = 1269/141 = 9

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit) = 9 cycles
Total setup/exit costs = 9 x 141 + 1(M0 conflict) = 1270
Saving per iteration: 12 -1 = 11 (4 for lp-ctl-oh, 7 for cs-load-oh)
cs-load saving: 7 x 1269 - 1270/2 = 8248
lp-ctl saving: 4 x 1128 + 3 x 141 - 1270/2 = 4300
Total saving: 8248 + 4300 = 12548

(ii) CVA-Only executions

Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit) + 1(t_p) + 0(t_s) -1 = 9 cycles
Execution time = 9 x 141 + 1269 = 2538
CVA saving = 15087 - 2538 = 12549

(iii) CVA/PVA executions

Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core during loop executions: 10 x 1128 + 9 x 141 = 12549
PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 141 + 1 x 1269 = 2256
CVA-only executions:
IReq from core during CVA executions: 7 (setup code/vector inst.) x 141 = 987

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 141 + 1 (essential inst.) = 988

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 7 (setup code/vector inst.) x 141 = 987
CVA/PVA executions: 7 (setup code/vector inst.) x 141 = 987

A.4.2 Critical Loop 2

Table A.4.3: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000a62 950e //; stw r5,(r14) 00000a64 203e //; addi r14,4 00000a66 01b6 //; decne r6 00000a68 e7fc //; bt 0x00000a62	PVA @S,#1 cs-store: mov r5, r5[a]	CVA mov r5, @S;

a. Essential instruction

Table A.4.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000a62 00000a68	target bt	33 33	- 00000a62	- 32 (97.0)	- 1 (3.03)

This loop performs vector initialization, C[i] = r5, for some scalar r5.
Estimated execution cycles = 6 x 33 = 198
Average number of iterations per invocation = 33

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup and exit costs = 8 x 1 + 1(M0 conflict) = 9
cs-load saving: 0
cs-store saving: 2 x 33 - 9/2 = 62
lp-ctl saving: 3 x 33 - 9/2 = 95
Total saving = 62 + 95 = 157

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) +
$+ 1(t_p) + 0(t_s) - 1 = 8$
Execution time = 8 x 1 + 33 = 41

CVA saving = 198 - 41 = 157

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 5 x 32 + 4 x 1 = 164
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 1 + 1 x 32 = 38
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 1 = 6

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 1 + 1 (essential inst.) = 7

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 7 (setup code/vector inst.) x 1 = 7
CVA/PVA executions: 7 (setup code/vector inst.) x 1 = 7

A.4.3 Summary

Total execution cycles: 586643
Total execution cycles in loops: 15087 + 198 = 15285 or 2.61% of total execution time.
Average number of iterations per invocation = (15087 x 9 + 33 x 198)/15285 = 9.31

(i) PVA-Only executions
Total cs-load saving = 8248
Total cs-store saving = 62
Total lp-ctl saving = 4300 + 95 = 4395
Total cycle saving: 12548+ 157 = 12705
Speedup during loop executions = 15285/ (15285 - 12706) = 5.929 (perf. imp. = 0.8313)
Overall speedup: 586643/(586643-12706) = 1.022 (perf. imp. = 0.0215)

Table A.4.5: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	4395	0.0075
cs-load-oh	8248	0.0141
cs-store-oh	62	0.0001
Total	12705	0.0217

(ii) CVA-Only executions
CVA-only with no early termination
Total saving: 157 (loop2)
Speedup during loop executions = 15285/ (15285 -157) = 1.010 (perf. imp. = 0.0099)

Overall speedup: 586643/(586643-157) = 1.000 (perf. imp. = 0.0000)

CVA-only with possible early termination
Total saving: 12550 + 157 = 12707
Speedup during loop executions = 15285/ (15285 - 12706) = 5.929 (perf. imp. = 0.8313)
Overall speedup: 586643/(586643-12706) = 1.022 (perf. imp. = 0.0215)

(iii) CVA/PVA executions
Total saving: 12550(CVA or PVA for loop2) + 157(CVA or PVA for loop2) = 12707
Speedup during loop executions = 15285/ (15285 - 12706) = 5.929 (perf. imp. = 0.8313)
Overall speedup: 586643/(586643-12706) = 1.022 (perf. imp. = 0.0215)

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core overall: 519037
IReq from core during loop executions: 12549 + 164 = 12713

PVA-only executions:
IReq from core reduced = 12713 - (2256 + 38) = 10419
Normalized IReq from core during PVA executions = (2256 + 28)/12713 = 0.1804
Normalized IReq from core overall = (519037 - 10419)/519037 = 0.9799

CVA-only executions:
IReq from core reduced = 12713 - (987 + 6) = 11720
Normalized IReq from core during CVA executions = (987 + 6)/12713 = 0.0781
Normalized IReq from core overall = (519037 - 11720)/519037 = 0.9774

CVA/PVA executions:
IReq from core reduced = 12713 - (987 + 6) = 11720
Normalized IReq from core during CVA/PVA executions = (987 + 6)/12713 = 0.0781
Normalized IReq from core overall = (519037 - 11720)/519037 = 0.9774

Table A.4.6: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.1804	0.0781	0.0781	0.9799	0.9774	0.9774

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory reduced = 12713 - (988 + 7) = 11718
Normalized Ifetch from memory during PVA executions: (988+7)/12713 = 0.0783
Normalized Ifetch from memory overall = (519037 - 11718)/519037 = 0.9774

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:
IReq from core = Ifetch from memory

CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.4.7: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.0781	0.0781	0.0781	0.0781	0.9774	0.9774	0.9774	0.9774

A.5 Benchmark “engine”

A.5.1 Critical Loop 1

Table A.5.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
000002d2 a76d //; ldb r7,(r13,6) 000002d4 205d //; addi r13,6 000002d6 0d27 //; cmplt r7,r2 000002d8 e7fc //; bt 0x000002d2	PVA @L0, ct=0,#1 cmplt r0, r2[a]	mov r5, r2 CVA cmplt.ct=0 @L0, r5;

a. Essential instruction

Table A.5.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000002d2 000002d8	target bt	16874 16874	- 000002d2	- 11995 (71.1)	- 4879 (28.9)

This loop reads a vector A sequentially and finds the first i that satisfies: A[i] < r2, for some scalar r2. Vector A is not sorted.

Estimated execution cycles = 6 x 11995 + 5 x 4879 = 96365

Average number of iterations per invocation = 16874/4879 = 3.46

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1

Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8

Total setup/exit costs = 8 x 4879 + 1(M0 conflict) = 39033

cs-load saving: 3 x 16874 - 39033/2 = 31106

lp-ctl saving: 2 x 11995 + 1 x 4879 - 39033/2 = 9353

Total saving = 31106 + 9353 = 40459

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit)
+ 1(t_p) + 0(t_s) - 1 + 1(extra “move” inst.) = 9 cycles

Execution time = 4879 x 9 + 16874 = 60785

CVA-only saivng = 96365 - 60785 = 35580

(iii) CVA/PVA executions

Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:

IReq from core during loop executions: 5 x 11995 + 4 x 4879 = 79491

PVA-only executions:

IReq from core during PVA executions: 6 (setup code/vector inst.) x 4879 + 1 x 16874 = 46148

CVA-only executions:

IReq from core during CVA executions: 6 (setup code/vector inst.) x 4879 = 29274

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 4879 + 1(essential inst.) = 29275

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 6 (setup code/vector inst.) x 4879 = 29274
CVA/PVA executions: 6 (setup code/vector inst.) x 4879 = 29274

A.5.2 Critical Loop 2

Table A.5.3: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
0000030e a76c //; ldb r7,(r12,6) 00000310 205c //; addi r12,6 00000312 0d27 //; cmplt r7,r2 00000314 e7fc //; bt 0x0000030e	PVA @L0, ct=0, #1 cmplt r0, r2[a]	mov r5, r2 CVA cmplt.ct=0 @L0, r5;

a. Essential instruction

Table A.5.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
0000030e 00000314	target bt	17309 17309	- 0000030e	- 12430 (71.8)	- 4879 (28.2)

This loop reads a vector A sequentially and finds the first i that satisfies: A[i] < r2, for some scalar r2. Vector A is not sorted.

Estimated execution cycles = 6 x 12430 + 5 x 4879 = 98975

Average number of iterations per invocation = 17309/4879 = 3.56

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8
Total setup/exit costs = 8 x 4879 - 1(M0 conflict) = 39033
cs-load saving: 3 x 17309 - 39033/2 = 32411
lp-ctl saving: 2 x 12430 + 1 x 4879 - 39033/2 = 10223
Total saving = 42634

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit)
+ 1(t_p) + 0(t_s) - 1 + 1(extra “move” inst.) = 9 cycles
Execution time = 4879 x 9 + 17309 = 61220
CVA-only saivng = 98975 - 61220 = 37755

(iii) CVA/PVA executions

Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 5 x 12430 + 4 x 4879 = 81666
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 4879 + 1 x 17309 = 46583
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 4879 = 29274

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 4879 + 1(essential inst.) = 29275

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 6 (setup code/vector inst.) x 4879 = 29274
CVA/PVA executions: 6 (setup code/vector inst.) x 4879 = 29274

A.4.3 Summary

Total execution cycles: 2000000
Total execution cycles in loops: 96365 + 98975 = 195340 or 9.77% of total execution time.
Average number of iterations per invocation = (3.46 x 96365 + 3.56 x 98975)/195340 = 3.51

(i) PVA-Only executions
cs-load saving = 31106 + 32411 = 63517
lp-ctl saving = 9353 + 10223 = 19576
Total cycle saving: 63517 + 19576 = 83093
Speedup during loop executions = 195340 / (195340 - 83093) = 1.740 (perf. imp. = 0.4253)
Overall speedup: 2000000/(2000000 - 83093) = 1.043 (perf. imp. = 0.0412)

Table A.5.5: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	19576	0.0098
cs-load-oh	63517	0.0318
cs-store-oh	0	0
Total	83093	0.0416

(ii) CVA-Only executions
CVA-only with no early termination
Total saving: 0
Speedup during loop executions = 1 (perf. imp. = 0.0000)
Overall speedup: 1 (perf. imp. = 0.0000)

CVA-only with possible early termination
Total cycle saving:35580 (loop 1) + 37755 (loop 2) = 73335
Speedup during loop executions = 195340 / (195340 - 73335) = 1.601(perf. imp. = 0.3754)
Overall speedup: 2000000/(2000000 - 73335) = 1.038 (perf. imp. = 0.0366)

(iii) CVA/PVA executions

Total cycle saving:40459 (PVA for loop 1) + 42634 (PVA for loop 2) = 83093
Speedup during loop executions = 195340 / (195340 - 83093) = 1.740 (perf. imp. = 0.4253)
Overall speedup: 2000000/(2000000 - 83093) = 1.043 (perf. imp. = 0.0412)

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core overall: 1058154
IReq from core during loop executions: 79491 + 81666 = 161157

PVA-only executions:
IReq from core reduced = 161157 - (46148 + 46583) = 68426
Normalized IReq from core during PVA executions = (46148 + 46583)/161157 = 0.5754
Normalized IReq from core overall = (1058154 - 68426)/1058154 = 0.9353

CVA-only executions:
IReq from core reduced = 161157 - (29274 + 29274) = 102609
Normalized IReq from core during CVA executions = (29274 + 29274)/161157 = 0.3633
Normalized IReq from core overall = (1058154 - 102609)/1058154 = 0.9030

CVA/PVA executions:
IReq from core reduced = 161157 - (29274 + 29274) = 102609
Normalized IReq from core during CVA/PVA executions = (29274 + 29274)/161157 = 0.3633
Normalized IReq from core overall = (1058154 - 102609)/1058154 = 0.9030

Table A.5.6: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.5754	0.3633	0.3633	0.9353	0.9030	0.9030

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory reduced = 161157 - (29275 + 29275) = 102607
Normalized Ifetch from memory during PVA executions: (29275+29275)/161157 = 0.3633
Normalized Ifetch from memory overall = (1058154 - 102607)/1058154 = 0.9030

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only and CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.5.7: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.3633	0.3633	0.3633	0.3633	0.9030	0.9030	0.9030	0.9030

A.6 Benchmark "fir_int"

A.6.1 Critical Loop 1

Table A.6.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

000002f8 243e //; subi r14,4
000002fa 8607 //; ldw r6,(r7)
000002fc 851e //; ldw r5,(r14,4)
000002fe 01b1 //; decne r1
00000300 0356 //; mul r6,r5(2 cycle)
00000302 1c64 //; add r4,r6
00000304 2037 //; addi r7,4
00000306 e7f8 //; bt 0x000002f8
	PVA @L0,@L1, #3
	mov r5, r0[a]
	mul r5, r1[a]
	add r4, r5
		CVA mul @L0, @L1, @P,
		add r3, @P, r3;

[a] Essential instructions

Table A.6.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000002f8 00000306	target bt	70482 70482	- 000002f8	- 68382 (97)	- 2100 (2.98)

This loop computes an inner product described by $\sum_i$ A[i]*B[i].

Estimated execution cycles in loops = 12 x 68382 + 11 x 2100 = 843684 or 69.74%
Average number of iterations per invocation = 70482/2100 = 33.6

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit) = 9 cycles
Total setup/exit costs = 9 x 2100 + 1(M0 conflict) = 18901
Cycles per iteration: 13
Cycle saving per iteration = 8 (5 for cs-laod; 3 for lp-ctl)
cs-load saving: 5 x 70482 - 18901/2 = 342960
lp-ctl saving: 3 x 68382 + 2 x 2100 - 18901/2 = 199896
Total saving = 342960 + 199896 = 542856
Speedup during loop executions = 843684 / (843684 - 542856) = 2.805 (perf. imp. = 0.6435)
Overall speedup = 1209720/(1209720-542856) = 1.814 (perf. imp. = 0.4487)

Table A.6.3: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	199896	0.1652
cs-load-oh	342960	0.2835
cs-store-oh	0	0.0000
Total	542856	0.4487

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit)
+ $t_p(2)$ + $t_s(1)$ - 1 = 11 cycles
Execution time = 11 x 2100 + 70482 = 93582
CVA-only saving: 843684 - 93582 = 750102
Speedup during loop executions = 843684/ (843684 - 750102) = 9.015 (perf. imp. = 0.8891)
Overall speedup: 1209720/(1209720-750102) = 2.632 (perf. imp. = 0.6201)

(iii) CVA/PVA executions

Total saving = 750102 (use CVA)
Speedup during loop executions = 843684/ (843684 - 750102) = 9.015
Overall speedup: 1209720/(1209720-750102) = 2.632

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core overall: 705966
IReq from core during loop executions: 9 x 68382 + 8 x 2100 = 632238

PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 2100 + 3 x 70482 = 226146
IReq from core reduced = 632238 - 226146 = 406092
Normalized IReq from core during PVA executions = 226146/632238 = 0.3577
Normalized IReq from core overall = (705966 - 406092)/705966 = 0.4248

CVA-only executions:
IReq from core during CVA executions: 7 (setup code/vector inst.) x 2100 = 14700
IReq from core reduced = 632238 - 14700 = 617538
Normalized IReq from core during CVA executions = 14700/632238 = 0.0233
Normalized IReq from core overall = (705966 - 617538)/705966 = 0.1253

CVA/PVA executions:
IReq from core during CVA/PVA executions: 7 (setup code/vector inst.) x 2100 = 14700
IReq from core reduced = 632238 - 14700 = 617538
Normalized IReq from core during CVA/PVA executions = 14700/632238 = 0.0233
Normalized IReq from core overall = (705966 - 617538)/705966 = 0.1253

Table A.6.4: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.3577	0.0233	0.0233	0.4248	0.1253	0.1253

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions = 7 (setup code/vector inst.) x 2100 + 2 (essential inst.)
+ 1 x 70482 (non-essential inst.) = 85184
Ifetch from memory reduced = 632238 - 85184 = 547054
Normalized Ifetch from memory during PVA executions: 85184/632238 = 0.1347

Normalized Ifetch from memory overall = (705966 - 547054)/705966 = 0.2251

PVA-only with all inst. caching:
Ifetch from memory during PVA executions = 7 (setup code/vector inst.) x 2100 + 2 (essential inst.)
+ 1 x 4 (non-essential inst.) = 14706
Ifetch from memory reduced = 632238 - 14706 = 617532
Normalized Ifetch from memory during PVA executions: 14706/632238 = 0.0233
Normalized Ifetch from memory overall = (705966 - 617532)/705966 = 0.1253

CVA-only executions:
IReq from core = Ifetch from memory

CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.6.5: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.1347	0.0233	0.0233	0.0233	0.2251	0.1253	0.1253	0.1253

A.7 Benchmark “g3fax”

A.7.1 Critical Loop 1

Table A.7.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000316 01b3 //; decne r3 00000318 a602 //; ldb r6,(r2) 0000031a 2002 //; addi r2,1 0000031c 1c64 //; add r4,r6 0000031e e7fb //; bt 0x00000316	PVA @L0, #1 add r4, r0[a]	CVA mov @L0, @P, add r3, @P, r3;

a. Essential instruction

Table A.7.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000316 0000031e	target bt	87538 87538	- 000002d2	- 87488 (99,9)	- 50 (0.0571)

This loop performs a vector sum, described by $\sum_i$ A[i].

Estimated execution cycles = 7 x 87488 + 6 x 50 = 612716
Average number of iterations per invocation = 87538/50 = 1750.8

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8
Total setup/exit costs = 8 x 50 + 1(M0 conflict) = 401
cs-load saving: 3 x 87538 - 401/2 = 262414
lp-ctl saving: 3 x 87488 + 2 x 50 - 401/2 = 262364
PVA-only saving: 262414 + 262364 = 524778

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit)
+ 1(t_p) + 1(t_s)-1 = 9 cycles
Execution time = 9 x 50 +87538 = 87988
CVA-only saving: 612716 - 87988 = 524728

(iii) CVA/PVA executions

Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core during loop executions: 6 x 87488 + 5 x 50 = 525178
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 50 + 1 x 87538 = 87838
CVA-only executions:

IReq from core during CVA executions: 6 (setup code/vector inst.) x 50 = 300

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 50 + 1(essential inst.) = 301

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 6 (setup code/vector inst.) x 50 = 300
CVA/PVA executions: 6 (setup code/vector inst.) x 50 = 300

A.7.2 Critical Loop 2

Table A.7.3: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
000003f2 ba0e //; stb r10,(r14) 000003f4 200e //; addi r14,1 000003f6 018d //; declt r13 000003f8 effb //; bf 0x000003f2	PVA @S, #1 cs-store: mov r10, r10[a]	mov r5, r10 CVA mov r5, @S

a. Essential instruction

Table A.7.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000003f2 000003f8	target bt	86717 86717	- 000003f2	- 82885 (95.6)	- 3832 (4.42)

This loop performs vector initialization, C[i] = r10, for some scalar r10.
Estimated execution cycles = 6 x 82885 + 5 x 3832 = 516470
Average number of iterations per invocation = 86717/3832 = 22.63

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup/exit costs = 8 x 3832 + 1(M0 conflicts) = 30657
cs-store saving: 2 x 86717 - 30657/2 = 158106
lp-ctl saving: 3 x 82885 + 2 x 3832 - 30657/2 = 240991
Total saving = 158106 + 240991 = 399097

(ii) CVA-Only executions
Since t_p=1 and t_s=0, CVA saving = PVA saving = 399097

(iii) CVA/PVA executions
Same as CVA-only or PVA-only executions

(iv) CVA Executions Using Various Sizes of
N.A. (TM not used).

(v) IReq From Core
Base machinne:

IReq from core during loop executions: 5 x 82885 + 4 x 3832 = 429753
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 3832 + 1 x 86717 = 109709
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 3832 = 22992

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 3832 + 1(essential inst.) = 22993

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.
CVA-only executions: 6 (setup code/vector inst.) x 3832 = 22992
CVA/PVA executions: 6 (setup code/vector inst.) x 3832 = 22992

A.4.3 Summary

Total execution cycles: 2000000
Total execution cycles in loops: 612716 + 516470 = 1129186 or 56.65% of total execution time.
Average number of iterations per invocation = (1750.8 x 612716 + 22.63 x 516470)/1129186
= 960.4

(i) PVA-Only executions
cs-load saving: 262414
cs-store saving: 158106
lp-ctl saving: 262364 + 240991 = 503355
Total cycle saving: 262414 + 158106 + 503355 = 923875
Speedup during loop executions = 1129186/(1129186-923875) = 5.500 (perf. imp. = 0.8182)
Overall speedup: 2000000/(2000000-923875) = 1.859 (perf. imp. = 0.4621)

Table A.7.5: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	503355	0.2517
cs-load-oh	262414	0.1311
cs-store-oh	158106	0.0791
Total	923875	0.4619

(ii) CVA-Only executions
CVA-only saving: 524728(loop1) + 399097(loop2) = 923825
Speedup during loop executions = 1129186/(1129183-923825) = 5.499 (perf. imp. = 0.8182)
Overall speedup: 2000000/(2000000-923825) = 1.858 (perf. imp. = 0.4621)

(iii) CVA/PVA executions
CVA/PVA saving: 524778(PVA for loop1) + 399097(CVA or PVA for loop2) = 923857
Speedup during loop executions = 1129183/(1129183-923857) = 5.500 (perf. imp. = 0.8182)
Overall speedup: 2000000/(2000000-923857) = 1.859 (perf. imp. = 0.4621)

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core overall: 1681130
IReq from core during loop executions: 525178 + 429753 = 954931

PVA-only executions:
IReq from core reduced = 954931 - (87838 + 109709) = 757384
Normalized IReq from core during PVA executions = (87838 + 109709)/954931 = 0.2069
Normalized IReq from core overall = (1681130 - 757384)/1681130 = 0.5495

CVA-only executions:
IReq from core reduced = 954931 - (300 +22992) = 931639
Normalized IReq from core during CVA executions = (300 +22992)/954931 = 0.0244
Normalized IReq from core overall = (1681130 - 931639)/1681130 = 0.4458

CVA/PVA executions:
IReq from core reduced = 954931 - (300 +22992) = 931639
Normalized IReq from core during CVA/PVA executions = (300 +22992)/954931 = 0.0244
Normalized IReq from core overall = (1681130 - 931639)/1681130 = 0.4458

Table A.7.6: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.2069	0.0244	0.0244	0.5495	0.4458	0.4458

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory reduced = 954931 - (301 + 22993) = 931637
Normalized Ifetch from memory during PVA executions: (301 + 22993)/954931 = 0.0244
Normalized Ifetch from memory overall = (1681130 - 931637)/1681130 = 0.4458

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:
IReq from core = Ifetch from memory

CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.7.7: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.0244	0.0244	0.0244	0.0244	0.4458	0.4458	0.4458	0.4458

A.8 Benchmark “g721”

A.8.1 Critical Loop 1

Table A.8.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
0000088a 01b9 //; decne r9 0000088c c70c //; ldh r7,(r12) 0000088e 241c //; subi r12,2 00000890 d702 //; sth r7,(r2) 00000892 2412 //; subi r2,2 00000894 e7fa //; bt 0x0000088a	PVA @L0, @S, #1 cs-store: mov r0, r0[a]	CVA mov @L0, @S;

a. Essential instruction

Table A.8.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
0000088a 00000894	target bt	990 990	- 00000304	- 792 (80)	- 198 (20)

This loop performs a vector move C[i] = A[i].
Estimated execution cycles = 9 x 792 + 8 x 198 = 8712 or 2.87% execution time.
Average number of iterations per invocation = 990/198 = 5

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2)
+ 1(exit) = 9 cycles
Total setup/exit costs = 9 x 198 + 1(M0 conflict) = 1783
Cycle saving per iteration = 9 -1(extra “mov”)=8
cs-load saving: 3 x 990 - 1783/3 = 2376
cs-store saving: 2 x 990 - 1783/3 = 1386
lp-ctl saving: 3 x 792 + 2 x 198 - 1783/3 = 2178
Total saving = 2376 + 1386 + 2178 = 5940
Speedup during loop executions = 8712/(8712 - 5940) = 3.143 (perf. imp. = 0.6818)
Overall speedup = 303885/(303885-5940) = 1.020 (perf. imp. = 0.0196)

Table A.8.3: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	2178	0.0072
cs-load-oh	2376	0.0078
cs-store-oh	1386	0.0046
Total	5940	0.0196

(ii) CVA-Only executions

Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2)

$$+ 1(\text{exit}) + 1(t_p) + 0(t_s) - 1 = 9 \text{ cycles}$$

Execution time = 9 x 198 + 990 = 2772
CVA saving = 8712 - 2772 = 5940
Speedup during loop executions = 8712/(8712 - 5940) = 3.143 (perf. imp. = 0.6818)
Overall speedup = 303885/(303885-5940) = 1.020 (perf. imp. = 0.0196)

(iii) CVA/PVA executions

Total saving = 5942 (CVA or PVA for loop1)
Speedup during loop executions = 8712/(8712 - 5942) = 3.143 (perf. imp. = 0.6818)
Overall speedup = 303885/(303885-5942) = 1.020 (perf. imp. = 0.0196)

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core overall: 256025
IReq from core during loop executions: 7 x 792 + 6 x 198 = 6732

PVA-only executions:
IReq from core during PVA executions = 7 (setup code and vector inst.) x 198 + 990 = 2376
IReq from core reduced = 6732 - 2376 = 4356
Normalized IReq from core during PVA executions = 2376/6732 = 0.3584
Normalized IReq from core overall = (256025 - 4356)/256025 = 0.9830

CVA-only executions:
IReq from core during CVA executions = 7 (setup code and vector inst.) x 198 = 1386
IReq from core reduced = 6732 - 1386 = 5346
Normalized IReq from core during CVA executions = 1386/6732 = 0.2059
Normalized IReq from core overall = (256025 - 5346)/256025 = 0.9791

CVA/PVA executions:
IReq from core during CVA/PVA executions = 7 (setup code and vector inst.) x 198 = 1386
IReq from core reduced = 6732 - 1386 = 5346
Normalized IReq from core during CVA/PVA executions = 1386/6732 = 0.2059
Normalized IReq from core overall = (256025 - 5346)/256025 = 0.9791

Table A.8.4: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.3584	0.2059	0.2059	0.9830	0.9791	0.9791

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7 (setup code and vector inst.)x198 + 1(essential inst) = 1387
Ifetch from memory reduced = 6732 - 1387 = 5345
Normalized Ifetch from memory during PVA executions: 1387/6732 = 0.2059
Normalized Ifetch from memory overall = (256025 - 5345)/256025 = 0.9791

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:

IReq from core = Ifetch from memory

CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.8.5: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.2059	0.2059	0.2059	0.2059	0.9791	0.9791	0.9791	0.9791

A.9 Benchmark “jpeg”

A.9.1 Critical Loop 1

Table A.9.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
000003cc 9105 //; stw r1,(r5) 000003ce 2035 //; addi r5,4 000003d0 01b6 //; decne r6 000003d2 e7fb //; bt 0x000003cc	PVA @S, #1 cs-store: mov r1, r1[a]	mov r5, r1 CVA mov r5, @S;

a. Essential instruction

Table A.9.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000003cc 000003d2	target bt	2625 2625	- 000003cc	- 1752 (66.7)	- 873 (33.3)

This loop performs vector initialization, C[i] = r1, for some scalar r1.
Estimated execution cycles = 7 x 1752 + 6 x 873 = 17502
Average number of iterations per invocation = 2625/873 = 3.01

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup/exit costs = 8 x 873 + 1(M0 conflict) = 6985
cs-store saving: 2 x 2625 - 6985/2 = 1758
lp-ctl saving: 3 x 1752 + 2 x 873 - 6985/2 = 3510
Total saving = 1758 + 3510 = 5268

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit)
+ 1(t_p) + 0(t_s) - 1 + 1(extra “mov” inst.) = 9 cycles
Execution time = 9 x 873 + 2625 = 10482
CVA-only saving: 17502 - 10482 = 7020

(iii) CVA/PVA executions

Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core during loop executions: 5 x 1752 + 4 x 873 = 12252
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 873 + 1 x 2625 = 7863
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 873 = 5238

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 873 + 1(essential inst.) = 5239

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 6 (setup code/vector inst.) x 873 = 5238
CVA/PVA executions: 6 (setup code/vector inst.) x 873 = 5238

A.9.2 Critical Loop 2

Table A.9.3: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
0000042c 960e //; stw r6,(r14) 0000042e 203e //; addi r14,4 00000430 01bd //; decne r13 00000432 e7fb //; bt 0x0000042c	PVA @S, #1 cs-store: mov r6, r6[a]	mov r5, r6 CVA mov r5, @S;

a. Essential instruction

Table A.9.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
0000042c 00000432	target bt	32595 32595	- 0000042c	- 31995 (98.2)	- 600 (1.8)

This loop performs vector initialization, C[i] = r6, for some scalar r6.
Estimated execution cycles = 6 x 31995 + 5 x 600 = 194970
Average number of iterations per invocation = 32595/600 = 54.325

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 0
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup/exit costs = 8 x 600 + 1(M0 conflict) = 4801
cs-store saving: 2 x 32595 - 4801/2 = 62790
lp-ctl saving: 3 x 31995 + 2 x 600 - 4801/2 = 94785
Total saving = 62790 + 94785 = 157575

(ii) CVA-Only executions

Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) + 1(t_p) + 0(t_s) - 1 = 8
Execution time = 8 x 600 + 32595 = 37395
CVA-only saving: 194970 - 37395 = 157575

(iii) CVA/PVA executions

Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:

IReq from core during loop executions: 5 x 31995 + 4 x 600 = 162375
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 600 + 1 x 32595 = 36195
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 600 = 3600

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 600 + 1(essential inst.) = 3601

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 6 (setup code/vector inst.) x 600 = 3600
CVA/PVA executions: 6 (setup code/vector inst.) x 600 = 3600

A.9.3 Critical Loop 3

Table A.9.5: Vertorizing Critical Loop 3

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000442 a604 //; ldb r6,(r4) 00000444 12b7 //; mov r7,r11 00000446 1d67 //; ixh r7,r6 00000448 860a //; ldw r6,(r10) 0000044a 01bc //; decne r12 0000044c d607 //; sth r6,(r7) // not cs 0000044e 2004 //; addi r4,1 00000450 203a //; addi r10,4 00000452 e7f7 //; bt 0x00000442	PVA @L0,@L1, #3 mov r7,r11[a] ixh r7,r0[a] sth r1,(r7)[a]	Not vectorizable.

[a] Essential instructions

Table A.9.6: Profile For Critical Loop 3

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000442 00000452	target bt	18600 18600	- 00000442	- 18000 (96.8)	- 600 (3.2)

Estimated execution cycles = 13 x 18000 + 12 x 600 = 241200
Average number of iterations per invocation = 18600/600 = 31

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 2
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1)
+ 1(exit) = 9 cycles
Total setup/exit costs = 9 x 600 + 2(M0 conflicts) = 5402
Cycle saving per iteration = 13 - 4 = 9 (3 for lp-ctl, 6 for cs-load)
cs-load saving: 6 x 18600 - 5402/2 = 108899
lp-ctl saving: 3 x 18000 + 2 x 600 - 5402/2 = 52499
PVA-only saving = 108899 + 52499 = 161398

(ii) CVA-Only executions
N.A.

(iii) CVA/PVA executions
Same as PVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A.

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 10 x 18000 + 9 x 600 = 185400
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 600 + 3 x 18000 = 57600
CVA-only executions: N.A.
CVA/PVA executions:
IReq from core during CVA/PVA executions: 6 (setup code/vector inst.)x600 + 3x18000 = 57600

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 600 + 3(essential inst.) = 3603

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: N.A.

CVA/PVA executions: same as PVA-only executions.

A.9.4 Critical Loop 4

Table A.9.7: Vertorizing Critical Loop 4

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA/PVA Construct

0000045e 240d //; subi r13,1
00000460 a61d //; ldb r6,(r13,1)
00000462 3c16 //; lsli r6,1
00000464 14b6 //; rsub r6,r11
00000466 1c76 //; add r6,r7
00000468 870c //; ldw r7,(r12)
0000046a 01ba //; decne r10
0000046c d706 //; sth r7,(r6) //not cs
0000047e 203c //; addi r12,4
00000470 e7f4 //; bt 0x0000045e
	PVA @L0,@L1, #6
	mov r6, r0
	lsli r6,1
	rsub r6,r11[a]
	add r6,r7[a]
	sth r1,(r6)[a]
		CVA lsli @L0, 1, @P,
		rsub @P, r11, @S;
		
		PVA @L0, #3
		mov r6, r0[a]
		add r6, r7[a]
		sth r1, (r6)[a]

a Essential instructions

Table A.9.8: Profile For Critical Loop 4

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
0000045e 00000470	target bt	18600 18600	- 0000045e	- 18000 (96.8)	- 600 (3.2)

Estimated execution cycles = 14 x 18000 + 13 x 600 = 259800
Average number of iterations per invocation = 18600/600 = 31

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 2
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(exit) = 9
Total setup/exit costs = 9 x 600 + 2(M0 conflicts) = 5402
Saving per iteration = 14 - 6 = 8 (3 for lp-ctl, 5 for cs-load)
cs-load saving: 5 x 18600 - 5402/2 = 90299
lp-ctl saving: 3 x 18000 + 2 x 600 - 5402/2 = 52499
PVA-only saving: 90299 + 52499 = 142798

(ii) CVA-Only executions

N.A.

(iii) CVA/PVA executions

The temporary vector produced by the first CVA instruction can be stored in M0/M1. TM is not used.
CVA instruction (t_p=t_s=1):
Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 1(t_p) + 1(t_s) - 1
= 10 cycles
Eecution time = 10 x 600 + 18600 = 24600
PVA instruction:
Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8 cycles
Execution time = 8 x 600 + 4 x 18600 = 79200
CVA/PVA saving: 259800 - 24600 - 79200 = 156000

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core during loop executions: 11 x 18000 + 10 x 600 = 204000
PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 600 + 5 x 18600 = 97200
CVA-only executions: N.A.

CVA/PVA executions:
CVA instruction:
IReq from core during CVA executions: 7 (setup code/vector inst.) x 600 = 4200
PVA instruction:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 600 + 3 x 18600 = 59400
Total IReq from core: 4200 + 59400 = 63600

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 600 + 3(essential inst.)
+ 2 x 18600 (non-essential inst.) = 41403

PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 600 + 3(essential inst.)
+ 2 x 4 (non-essential inst.) = 4211

CVA-only executions: N.A.

CVA/PVA executions:
CVA instruction:
Ifetch from memory during CVA executions: 7 (setup code/vector inst.) x 600 = 4200

PVA instruction:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 600 + 3(essential inst.) = 3603
Total IReq from core: 4200 + 3603 = 7803

A.9.5 Critical Loop 5

Table A.9.9: Vertorizing Critical Loop 5

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

00000614 c702 //; ldh r7,(r2)
00000616 a603 //; ldb r6,(r3)
00000618 01b1 //; decne r1
0000061a 0367 //; mul r7,r6 (2 cycle)
0000061c d702 //; sth r7,(r2)
0000061e 2012 //; addi r2,2
00000620 2003 //; addi r3,1
00000622 e7f8 //; bt 0x00000614
	PVA @L0,@L1,@S, #2
	mov r3, r0[a]
	cs-store:
	mul r3, r1[a]
		CVA mul @L0,@L1,@S

a. Essential instruction

Table A.9.10: Profile For Critical Loop 5

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000614 00000622	target bt	18048 18048	- 00000614	- 17766 (98.4)	- 282 (1.6)

This loop performs a vector multiplication A[i] = A[i] * B[i]. Vector length is always 64.
Estimated execution cycles = 13 x 17766 + 12 x 282 = 234342
Average number of iterations per invocation = 18048/282 = 64

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 1
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) = 10
Total setup/exit costs = 10 x 282 + 1(M0 conflicts) = 2821
saving per iteration: 14 - 4 = 10 (3 for lp-ctl, 4 for cs-load, 3 for cs-store)
cs-load saving: 4 x 18048 - 2821/3 = 71252
cs-store saving: 3 x 18048 - 2821/3 = 53204
lp-ctl saving: 3 x 17766 + 2 x 282 - 2821/3 = 52922
PVA-only saving: 71252 + 53204 + 52922 = 177378

(ii) CVA-Only executions
Vector duplications is used for this loop (see Section 6.10 on page 108).
Overhead for vector duplications = 282 x 7 = 1974 cycles.
Vector length = 64 halfwords or 128 bytes. Strip-mining of TM not necessary.
Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) +2(tp) + 0(ts) - 1 = 11 cycles
Execution time = 1974(vector duplications) + 11 x 282 + 18048 = 23124
CVA-only saving: 234342 - 23124 = 211218

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM

No TM:
Execution time = 11 x 282 + 18048 x 2 = 39198
Total saving = 234342 - 39198 = 195144

TM is 64 bytes:
In each invocation, vector length, n = 64 (or 128 bytes). TM needs to be strip-mined.
Execution time per invocation: 24 + (64/32)x(43+12) + 64 = 198
Total execution time = 198 x 282 + 1974(vector duplications) = 57810 (> 39198). TM not used.

TM is 128 bytes or larger:
CVA-only saving: 234342 - 23124 = 211218

Table A.9.11: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	Using TM?	Total Exe. Time	CVA-only Cycle Saving	Speedups during executions of loop 5
0	N	39198	195144	5.978
64	N	39198	195144	5.978
128	Y	23124	211218	10.134
256	Y	23124	211218	10.134
512	Y	23124	211218	10.134
1024	Y	23124	211218	10.134

(v) IReq From Core

Base machinne:
IReq from core during loop executions: 9 x 17766 + 8 x 282 = 162150
PVA-only executions:
IReq from core during PVA executions: 8 (setup code/vector inst.) x 282 + 2 x 18048 = 38352
CVA-only executions:
IReq from core during CVA executions: 8 (setup code/vector inst.) x 282 = 2256

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 8 (setup code/vector inst.) x 282 + 2(essential inst.) = 2258

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: 8 (setup code/vector inst.) x 282 = 2256
CVA/PVA executions: 8 (setup code/vector inst.) x 282 = 2256

A.9.6 Summary

Total cycles in loops = 17502 + 194970 + 241200 +259800 + 234342 = 947814 or 49.02%
Average number of iterations per invocation = (17502 x 3.01 + 194970 x 5.49 + 241200 x 31 +
259800 x 31 + 234342 x 64.00)/947814 = 33.39

(i) PVA-Only executions

Total cs-load saving: 108899(loop3) + 90299(loop4) + 71252(loop5) = 270450
Total cs-store saving: 1758(loop1) + 62790(loop2) + 53204(loop5) = 117752
Total lp-ctl saving: 3510(loop1) + 94785(loop2) + 52499(loop3) + 52499(loop4)

+ 52922(loop5) = 256215
Total saving = 270450 + 117752 + 256215 = 644417
Speedup duirng loop executions: 947814/(947814-644417) = 3.124 (perf. imp. = 0.6799)
Overall speedup: 2000000/(2000000-644417) = 1.475 (perf. imp. = 0.3220)

Table A.9.12: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	256215	0.1281
cs-load-oh	270450	0.1352
cs-store-oh	117752	0.0589
Total	644417	0.3222

(ii) CVA-Only executions
Total cycles in loops = 17502(loop1) + 227565(loop2) + 234342(loop5) = 479409
Total CVA-only saving: 7020(loop1) + 189570(loop2) + 211218(loop5) = 407808
Speedup duirng loop executions: 479409/(479409- 407808) = 6.696 (perf. imp. = 0.8506)
Overall speedup: 2000000/(2000000-407808) = 1.256 (perf. imp. = 0.2039)

(iii) CVA/PVA executions
Total saving: 7020(CVA for loop1) + 189570(CVA for loop2) + 161400(PVA for loop3)
+ 156000(combination of CVA and PVA) + 211218(CVA for loop5) = 725208
Speedup during loop executions = 947814/(947814-725208) = 4.258 (perf. imp. = 0.7651)
Overall speedup = 2000000/(2000000-725208) = 1.569 (perf. imp. = 0.3626)

(iv) CVA Executions Using Various Sizes of TM
No TM or TM is 64 bytes (not used).
CVA-only executions:
Total cycles in loops = 17502(loop1) + 227565(loop2) + 234342(loop5) = 479409
Total CVA-only saving: 7020(loop1) + 189570(loop2) + 195144(loop5) = 391734
Speedup duirng loop executions: 479409/(479409-391734) = 5.468 (perf. imp. = 0.8171)
Overall speedup: 2000000/(2000000-391734) = 1.244 (perf. imp. = 0.1959)

CVA/PVA executions:
Total saving: 7020(CVA for loop1) + 189570(CVA for loop2) + 161400(PVA for loop3)
+ 156000(combination of CVA and PVA) + 195144(CVA for loop5) = 709134
Speedup during loop executions = 947814/(947814-709134) = 3.971(perf. imp. = 0.7482)
Overall speedup = 2000000/(2000000-709134) = 1.5493 (perf. imp. = 0.3546)

Table A.9.13: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	CVA-Only Executions			CVA/PVA Executions		
	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup
0	391734	5.468	1.244	709134	3.971	1.549
64	391734	5.468	1.244	709134	3.971	1.549
128	407808	6.696	1.256	725208	4.258	1.569

Table A.9.13: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	CVA-Only Executions			CVA/PVA Executions		
	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup
256	407808	6.696	1.256	725208	4.258	1.569
512	407808	6.696	1.256	725208	4.258	1.569
1024	407808	6.696	1.256	725208	4.258	1.569

(v) IReq From Core

Base machinne:
IReq from core overall: 1528812
IReq from core during loop executions (all 5 loops):
12252 + 162375 + 185400 + 204000 + 162150=726177

PVA-only executions:
IReq from core reduced = 726177 - (7863+36195+57600+97200+38352) = 488967
Normalized IReq from core during PVA executions:
(7863+36195+57600+97200+38352)/726177=0.3267
Normalized IReq from core overall = (1528812 - 488967)/1528812 = 0.6802

CVA-only executions:
IReq from core during CVA executions: 12252(loop1)+162375(loop2)+162150(loop5)=336777
IReq from core reduced = 336777 - (5238(loop1) + 3600(loop2) + 2256(loop5))= 325683
Normalized IReq from core during CVA executions = (5238 + 3600 + 2256)/336777 = 0.0329
Normalized IReq from core overall = (1528812 - 325683)/1528812 = 0.7870

CVA/PVA executions:
IReq from core during CVA/PVA executions:
5238(loop2)+3600(loop2)+57600(loop3)+63600(loop4)+2256(loop5)=132294
IReq from core reduced = 726177 - 132294 = 593883
Normalized IReq from core during CVA/PVA executions = 132294/726177 = 0.1822
Normalized IReq from core overall = (1528812 - 593883)/1528812 = 0.6115

Table A.9.14: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.3267	0.0329	0.1822	0.6802	0.7870	0.6115

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during loop executions: 5239(loop1) + 3601(loop2) + 3603(loop3) + 41403(loop4)
+ 2258(loop5) = 56104
Ifetch from memory reduced = 726177 - 56104 = 670073
Normalized Ifetch from memory during PVA executions: 56104/726177 = 0.0773
Normalized Ifetch from memory overall = (1528812 - 670073)/1528812 = 0.5617

PVA-only with all inst. caching:

Ifetch from memory during loop executions: 5239(loop1) + 3601(loop2) + 3603(loop3) + 4211(loop4) + 2258(loop5) = 18901
Ifetch from memory reduced = 726177 - 18901 = 707276
Normalized Ifetch from memory during PVA executions: 18901/726177 = 0.0260
Normalized Ifetch from memory overall = (1528812 - 707276)/1528812 = 0.5374

CVA-only executions:
Ifetch from memory during loop executions: 5238(loop1) + 3600(loop2) + 2256(loop5) = 11094
Ifetch from memory reduced = 336777 - 11094 = 325683
Normalized Ifetch from memory during PVA executions: 11094/336777 = 0.0329
Normalized Ifetch from memory overall = (1528812 - 325683)/1528812 = 0.7870

CVA/PVA executions:
Ifetch from memory during loop executions: 5239(loop1) + 3600(loop2) + 3603(loop3) + 7803(loop4) + 2256(loop5) = 22500
Ifetch from memory reduced = 726177 - 22500 = 703677
Normalized Ifetch from memory during PVA executions: 22500/726177 = 0.0310
Normalized Ifetch from memory overall = (1528812 - 703677)/1528812 = 0.5397

Table A.9.15: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.0773	0.0260	0.0329	0.0310	0.5617	0.5374	0.7870	0.5397

A.10 Benchmark “map3d”

A.10.1 Critical Loop 1

Table A.10.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000458 c702 //; ldh r7,(r2) 0000045a 2012 //; addi r2,2 0000045c 0d73 //; cmplt r3,r7 0000045e e002 //; bt 0x00000464 00000460 01be //; decne r14 00000462 e7fa //; bt 0x00000458	PVA @L0, ct=0, #1 cmplt r3, r0[a]	mov r4, r3 CVA cmplt.ct=0 r4, @L0;

a. Essential instruction

Table B.10.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000458	target	96604	-	-	-
0000045e	bt	96604	00000464	7155 (7.41)	89449 (92.6)
00000462	bt	89449	00000458	89449 (100)	0 (0)

This loop performs reads in a vector A sequentially and find the first i that satisfies r3 < A[i]. Vector A is not sorted.
Estimated execution cycles = 6 x 7155 + 8 x 89449 = 758522 or 37.93% of total execution time.
Average number of iterations per invocation = 96604/7155 = 13.5

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8
Total setup/exit costs = 8 x 7155 + 1(M0 conflict) = 57241
cs-load saving: 3 x 96604 - 57241/2 = 261192
lp-ctl saving: 4 x 89449 + 2 x 7155 - 57241/2 = 343486
Total saving = 261192 +343486 = 604678
Speedup during loop executions = 758522/(758522- 604678) = 4.930 (perf. imp. = 0.7972)
Overall speedup = 2000000/(2000000-604678) = 1.433 (perf. imp. = 0.3022)

Table A.10.3: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	343486	0.1717
cs-load-oh	261192	0.1306
cs-store-oh	0	0
Total	604678	0.3023

(ii) CVA-Only executions

CVA-only with no early termination
Cycle saving: 0
Speedup during loop executions = 1 (perf. imp. = 0.0000)

Overall speedup = 1 (perf. imp. = 0.0000)

CVA-only with possible early termination
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) + 1(t_p)
+ 0(t_s) - 1 + 1(extra "mov" inst.) = 9 cycles
Execution time = 9 x 7155 + 96604 = 160999
CVA-only saving: 758522 - 160999 = 597523
Speedup during loop executions = 758522/(758522- 597523) = 4.711 (perf. imp. = 0.7877)
Overall speedup = 2000000/(2000000-597523) = 1.426 (perf. imp. = 0.2987)

(iii) CVA/PVA executions
CVA/PVA saving: 604678 (PVA for loop1)
Speedup during loop executions = 758522/(758522- 604678) = 4.930 (perf. imp. = 0.7972)
Overall speedup = 2000000/(2000000-604678) = 1.433 (perf. imp. = 0.3022)

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core overall: 1463233
IReq from core during loop executions: 7 x 89449 + 5 x 7155 = 661918

PVA-only executions:
IReq from core during PVA executions: 6 (setup code and vector inst.)x7155 + 1x96604 = 139534
IReq from core reduced = 661918 - 139534 = 522384
Normalized IReq from core during PVA executions = 139534/661918 = 0.2108
Normalized IReq from core overall = (1463233 - 522384)/1463233 = 0.6430

CVA-only executions:
IReq from core during CVA executions: 6 (setup code and vector inst.) x 7155 = 42930
IReq from core reduced = 661918 - 42930 = 618988
Normalized IReq from core during CVA executions = 42930/661918 = 0.0649
Normalized IReq from core overall = (1463233 - 618988)/1463233 = 0.5770

CVA/PVA executions:
IReq from core during CVA/PVA executions: 6 (setup code and vector inst.) x 7155 = 42930
IReq from core reduced = 661918 - 42930 = 618988
Normalized IReq from core during CVA/PVA executions = 42930/661918 = 0.0649
Normalized IReq from core overall = (1463233 - 618988)/1463233 = 0.5770

Table A.10.4: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.2108	0.0649	0.0649	0.6430	0.5770	0.5770

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6(setup code and vector inst.)x7155 + 1(essential inst)
= 42931
Ifetch from memory reduced = 661918 - 42931 = 618987

Normalized Ifetch from memory during PVA executions: 42931/661918 = 0.0649
Normalized Ifetch from memory overall = (1463233 - 618987)/1463233 = 0.5770

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:
IReq from core = Ifetch from memory

CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.10.5: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.0649	0.0649	0.0649	0.0649	0.5770	0.5770	0.5770	0.5770

A.11 Benchmark “pocsag”

A.11.1 Critical Loop 1

Table A.11.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Execution Cycles	Number Inst. Fetches	Using PVA Construct	Using CVA Construct
000002de 0e21 //; tst r1,r2 000002e0 e80f //; bf 0x000002f0 000002e2 2419 //; subi r9, 1 000002e4 a606 //; ldb r6,(r9) 000002e6 1767 //; xor r12,r6 000002e8 241a //; subi r10, 1 000002ea a606 //; ldb r6,(r10) 000002ec 1767 //; xor r13,r6 000002ee 3e12 //; lsri r2,1 000002f0 0183 //; declt r3 000002fa efec //; bf 0x000002de	4557 2x2163 + 2394 2394 2 x 2394 2394 2394 2 x 2394 2394 2394 4557 2x4410 + 147	4557 2x2163+2394 2394 2394 2394 2394 2394 2394 2397 4557 2x4410+147	mov r4, r1 PVA@L0,@L1,#5 tst r4,r2[a] bf CONT[a] xor r12,r0 xor r13,r1 CONT: lsri r2,1	Not Vectorizable.
Total	46347	41559	-	-

[a] Essential instructions

Table A.11.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000002de	target	4557	-	-	-
000002e0	bf	4557	00000300	2163 (47.5)	2394 (52.5)
000002f0	target	4557	-	-	-
000002fa	bf	4557	000002de	4410 (96.8)	147 (3.23)

Estimated execution cycles = 46347 (see Table A.11.1)
Average number of iterations per invocation = 4557/147 = 31

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) +1(R1)
+ 1(exit) + 1(extra “mov” inst.) = 10 cycles
Total setup/exit costs = 10 x 147 + 1(M0 conflic) = 1471
cs-load saving: 6 x 2394 - 1471/2 = 13629
lp-ctl saving: 3 x 4410 + 2 x 147 - 1471/2 = 12789

(ii) CVA-Only executions
N.A.

(iii) CVA/PVA executions
Same as PVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A.

(v) IReq From Core
Base machinne:
IReq from core during loop executions:41559 (see Table A.11.1)

PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 147 + 5 x 4557 = 23814
CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 147 + 2(essential inst.)
+ 3 x 4557 (non-essential inst.) = 14702

PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 7 (setup code/vector inst.) x 147 + 2(essential inst.)
+ 3 x 4 (non-essential inst.) = 1043

CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only with all inst. caching

A.2.2 Critical Loop 2

Table A.11.3: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

000003cc 1226 //; mov r6,r2
000003ce 0146 //; zextb r6
000003d0 1c16 //; add r6,r1
000003d2 a606 //; ldb r6,(r6) //not cs
000003d4 01b4 //; decne r4
000003d6 1c63 //; add r3,r6
000003d8 3e82 //; lsri r2,8
000003da e7f8 //; bt 0x000003cc
	PVA #6
	mov r6,r2
	zextb r6
	add r6,r1
	ldb r6,(r6)
	add r3,r6
	lsri r2,8
		Not Vectorizable.

Table A.11.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000356 0000036c	target bt	4044 4044	- 00000356	- 3033 (75)	- 1011 (25)

Estimated execution cycles = 10 x 3033 + 11 x 1011 = 41451
Average number of iterations per invocation = 4044 x 1011 = 4

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 0
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(exit) = 7 cycles
Total setup/exit costs = 7 x 1011 + 0(M0 conflict) = 7077
lp-ctl saving: 3 x 3033 + 2 x 1011 - 7077 = 4044

(ii) CVA-Only executions
N. A.

(iii) CVA/PVA executions
Same as PVA-only executions

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 9 x 3033 + 8 x 1011 = 35385
PVA-only executions:
IReq from core during PVA executions: 5 (setup code/vector inst.) x 1011 + 6 x 4044 = 29319
CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 5 (setup code/vector inst.) x 1011
+ 6 x 4044 (non-essential inst.) = 29319

PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 5 (setup code/vector inst.) x 1011
+ 6 x 4 (non-essential inst.) = 5079

CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only with all inst. caching.

A.2.3 Summary

Total number of cycles in loops: 46347 + 41451 = 87798 or 61.21%
Average number of iterations per invocation = (46347 x 31 + 41451 x 4)/87798 = 18.3

(i) PVA-Only executions
cs-load saving: 13629
lp-ctl Saving: 12789 + 4044 = 16833
Total saving: 13629 + 16833 = 30462
Speedup during loop executions = 87798/ (87798 - 30462) = 1.531 (perf. imp. = 0.3468)
Overall speedup: 143437/(143437 - 30462) = 1.270 (perf. imp. = 0.2126)

Table A.11.5: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	16833	0.1174
cs-load-oh	13629	0.0950
cs-store-oh	0	0
Total	30462	0.2124

(ii) CVA-Only executions
Total cycle saving: 0
Speedup during loop executions = 1 (perf. imp. = 0.000)
Overall speedup = 1 (perf. imp. = 0.000)

(iii) CVA/PVA executions
Total cycle saving (PVA for loop1 and loop2) = 30462
Speedup during loop executions = 87798/ (87798 - 30462) = 1.531 (perf. imp. = 0.3468)

Overall speedup: 143437/(143437 - 30462) = 1.270 (perf. imp. = 0.2126)

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core overall: 147202
IReq from core during loop executions: 41559+ 35385 = 76944

PVA-only executions:
IReq from core reduced = 76944 - (23814 + 29319) = 23811
Normalized IReq from core during PVA executions = (23814 + 29319)/76944 = 0.6905
Normalized IReq from core overall = (147202 - 23811)/147202 = 0.8382

CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only executions.

Table A.11.6: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.6905	1.0000	0.6905	0.8382	1.0000	0.8382

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory reduced = 76944 - (14702 + 29319) = 32923
Normalized Ifetch from memory during PVA executions: (14702 + 29319)/76944 = 0.5721
Normalized Ifetch from memory overall = (147202 - 32923)/147202 = 0.7763

PVA-only with all inst. caching:
Ifetch from memory reduced = 76944 - (1043 + 1011) = 74890
Normalized Ifetch from memory during PVA executions: (1043 + 1011)/76944 = 0.0267
Normalized Ifetch from memory overall = (147202 - 74890)/147202 = 0.4912

CVA-only executions: N.A.

CVA/PVA executions: same as PVA-only with all inst. caching.

Table A.11.7: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.5721	0.0267	1.0000	0.0267	0.7763	0.4912	1.0000	0.4912

A.12 Benchmark “servo”

Cycles in Loops = 0; Total cycle saving: 0; % cycle saving: 0
Speedup during loop executions: N.A.
Overall speedup = 1.

Table A.11.8: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	0	0
cs-load-oh	0	0
cs-store-oh	0	0
Total	0	0

Table A.11.9: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Table A.11.10: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

A.13 Benchmark “summin”

A.13.1 Critical Loop 1

Table A.13.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

000002c2 1217 //; mov r7,r1
000002c4 0317 //; mul r7,r1 (2 cycles)
000002c6 2001 //; addi r1,1
000002c8 3a17 //; asri r7,1
000002ca 0d31 //; cmplt r1,r3
000002cc 9702 //; stw r7,(r2)
000002ce 2032 //; addi r2,4
000002d0 e7f8 //; bt 0x000002c2
	PVA @S, #4
	mov r7,r1
	mul r7,r1
	addi r1,1[a]
	cs-store:
	asri r7,1[a]
		movi r4,1
		; T1[i] = i
		CVA mov r4, @P,
		add @P, r3, {r3,@S};
		. . . . . .
		; T2[i] = T1[i]^2
		CVA mul @L0, @L0, @S;
		. . . . . .
		; C[i] = asri(T2[i],1)
		CVA asri @L0, 1, @S;

a. Essential instruction

Table A.13.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000002c2 000002d0	target bf	1550 1550	- 000002c2	- 1519 (98)	- 31 (2)

This loop performs a vector operation described by C[i] = asri(i^2,1), for i=1,2,3...50.
Estimated execution cycles = 11 x 1519 + 10 x 31 = 17019
Average number of iterations per invocation = 1550/31 = 50
TM needs not be used (although it could be used for power reason).

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8 cycles
Total setup/exit costs = 8x 31 + 1(M0 conflict) = 249
Cycles per iteration: 5
Cycles saving per iteration = 11 - 5 = 6 (3 for cs-store, 3 for lp-ctl)
cs-store saving: 3 x 1550 - 249/2 = 4526
lp-ctl saving: 3 x 1519 + 2 x 31 - 249/2 = 4495
PVA saving: 4526 + 4495 = 9021

(ii) CVA-Only executions

TM needs not be used in order to retain the same performance level. The two temporary vectors, T1 and T2, can be stored in M0 and M1, respectively (or in M1 and M0, respectively).
First CVA instructions:
Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) + 1(t_p) + 1(t_s) - 1
+ 1(extra “mov” inst.) = 10
Second CVA instructions:
Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) + 2(t_p)
+ 0(t_s) - 1 = 11
Third CVA instructions:
Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 1(t_p) + 0(t_s) - 1

= 9
Total setup/exit costs = 10 + 11 + 9 = 30 cycles
Execution time = 30 x 31 + 1550 x 3 = 5580
CVA-only saving = 17019 - 5580 = 11439

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 9 x 1519 + 8 x 31 = 13919
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 31 + 5 x 1550 = 7936
CVA-only executions:
IReq from core during CVA executions: (6+8+7) (setup code/vector inst.) x 31 = 651
CVA/PVA executions: same as CVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 31 + 2(essential inst.)
+ 2 x 1550 (non-essential inst.) = 3288
PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 31 + 2(essential inst.)
+ 2 x 4 (non-essential inst.) = 196
CVA-only executions:
Ifetch from memory during CVA executions: (6+8+7) (setup code/vector inst.) x 31 = 651

CVA/PVA executions: same as CVA-only executions

A.13.2 Critical Loop 2

Table A.13.3: Vertorizing Critical Loop 2

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

000002e0 1251 //; mov r1,r5
000002e2 1c71 //; add r1,r7
000002e4 1276 //; mov r6,r7
000002e6 2007 //; addi r7,1
000002e8 3c26 //; lsli r6,2
000002ea 0d47 //; cmplt r7,r4
000002ec 0561 //; sub r1,r6
000002ee e7f8 //; bt 0x000002e0
	PVA #7
	mov r1,r5
	add r1,r7
	mov r6,r7
	addi r7,1
	lsli r6,2
	sub r1,r6
		Not vectorizable

Table A.13.4: Profile For Critical Loop 2

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
000002e0 000002ee	target bf	37200 37200	- 000002e0	- 36456 (98)	- 744 (2)

Estimated execution cycles = 9 x 36456 + 8 x 744 = 334056

Average number of iterations per invocation = 37200/744 = 50

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 0
Setup/exit costs per invocation = 1(CIR) + 2(instr. decode) + 1(exit) = 4 cycles
lp-ctl saving: 3 x 36456 + 2 x 744 - 4 x 744 = 107880
Total saving = 107880

(ii) CVA-Only executions
N.A.

(iii) CVA/PVA executions
Same as PVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 9 x 36456 + 8 x 744 = 334056
PVA-only executions:
IReq from core during PVA executions: 3 (setup code/vector inst.) x 744 + 6 x 37200 = 225432
CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 3 (setup code/vector inst.) x 744 +
+ 6 x 37200 (non-essential inst.) = 225432
PVA-only with all inst. caching:
Ifetch from memory during PVA executions: 3 (setup code/vector inst.) x 744
+ 6 x 4 (non-essential inst.) = 2256
CVA-only executions: N.A.

CVA/PVA executions: same as PVA-only with all inst. caching.

A.13.3 Critical Loop 3

Table A.13.5: Vertorizing Critical Loop 3

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

00000302 8402 //; ldw r4,(r2)
00000304 2032 //; addi r2,4
00000306 0d34 //; cmplt r4,r3
00000308 0243 //; movt r3,r4
0000030a 01b1 //; decne r1
0000030c e7fa //; bt 0x00000302
	PVA @L0, #2
	cmplt r0,r3[a]
	movt r3,r0[a]
		Not vectorizable

a. Essential instruction

Table A.13.6: Profile For Critical Loop 3

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000302 0000030c	target bf	70000 70000	- 00000302	- 60000 (85.7)	- 10000 (14.3)

This loop finds the minimum element of a vector A, min(A[i]).
Estimated execution cycles: 8 x 60000 + 7 x 10000 = 550000
Average number of iterations per invocation = 70000/10000 = 7

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 1
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(exit) = 8 cycles
Total setup/exit costs = 8x 10000 + 1(M0 conflict) = 80001
cs-load saving: 3 x 70000 - 80001/2 = 170000
lp-ctl saving: 3 x 60000 + 2 x 10000 - 80001/2 = 160000
Total saving: 170000 + 160000 = 330000

(ii) CVA-Only executions
N.A.

(iv) CVA Executions Using Various Sizes of TM
N.A.

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 7 x 60000 + 6 x 10000 = 480000
PVA-only executions:
IReq from core during PVA executions: 5 (setup code/vector inst.) x 10000 + 2 x 70000 = 190000
CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 5 (setup code/vector inst.) x 10000 + 2(essential inst.) = 50002

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions: N.A.
CVA/PVA executions: same as PVA-only with all inst. caching.

A.13.4 Critical Loop 4

Table A.13.7: Vertorizing Critical Loop 4

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
0000033c 910e //; stw r1,(r14) 0000033e 203e //; addi r14,4 00000340 01b2 //; decne r2 00000342 e7fc //; bt 0x0000033c	PVA @S, #1 cs-store: mov r1,r1[a]	CVA mov r1,@S

a. Essential instruction

Table A.13.8: Profile For Critical Loop 4

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
0000033c 00000342	target bf	37200 37200	- 0000033c	- 37169 (99.9)	- 31 (0.1)

This loop performs vector initialization, C[i] = r1, for some scalar r1.
Estimated execution cycles: 6 x 37169 + 5 x 31 = 223169
Average number of iterations per invocation = 37200/31 = 1200

(i) PVA-Only executions
Stall cycles due to M0 conflicts = 1
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8 cycles
Total setup/exit costs = 8 x 31 + 1 = 249
cs store saving: 2 x 37200 - 249/2 = 74276
lp-ctl saving: 3 x 37169 + 2 x 31 - 249/2 = 111445
PVA-only saving = 74276 + 111445 = 185721

(ii) CVA-Only executions
Since t_p=1, t_s=0, CVA saving = PVA saving = 185721

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 5 x 37169 + 4 x 31 = 185969
PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 31 + 1 x 37200 = 37386
CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 31 = 186
CVA/PVA executions: same as CVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6 (setup code/vector inst.) x 31 + 1(essential inst.) = 187

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.
CVA-only executions: 6 x 31 = 186
CVA/PVA executions: 6 x 31 = 186

A.13.5 Critical Loop 5

Table A.13.9: Vertorizing Critical Loop 5

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct

00000358 860d //; ldw r6,(r13)
0000035a 870e //; ldw r7,(r14)
0000035c 0336 //; mul r6,r3 (2 cycle)
0000035e 1c67 //; add r7,r6
00000360 01b5 //; decne r5
00000362 970e //; stw r7,(r14)
00000364 203e //; addi r14,4
00000366 203d //; addi r13,4
00000368 e7f7 //; bt 0x00000358
	PVA @L0,@L1,@S, #4
	mov r6, r3[a]
	mul r6, r0[a]
	cs-store:
	add r6, r1[a]
		mov r5, r3
		CVA mul @L0, r5, @P,
		add @P, @L1, @S;

a. Essential instruction

Table A.13.10: Profile For Critical Loop 5

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000358 00000368	target bt	36261 36261	- 00000358	- 34751 (95.8)	- 1510 (4.16)

This loop performs a vector operation described by, B[i] = r3 *A[i]+B[i], for some scalar r3.
Estimated execution cycles:14 x 34751 + 13 x 1510 = 506144
Number of iterations per invocation (fixed) = 24

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 2
Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R1) + 1(R2) + 1(exit) = 10
Total setup/exit costs = 10 x 1510 + 2 = 15102
Saving per iteration = 15 - 5 = 10 (3 for lp-ctl; 4 for cs-load; 3 for cs-store)
cs-load saving: 4 x 36261 - 15102/3 = 140010
cs-store saving: 3 x 36261 - 15102/3 = 103749
lp-ctl saving: 3 x 34751 + 2 x 1510 - 15102/3 = 102239
PVA-only savng: 140010 + 103749+ 102239 = 345998

(ii) CVA-Only executions

This loop is CVA vectorizable using vector duplication on vector B (see Section 6.10 on page 108).
Loop 5 is an inner loop of a doubly nested loop. The profile of this doubly nested loop is shown below.

Table A.13.11: More Profile For Critical Loop 5

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000346 00000358 00000368 0000036e	target target bt bt	1510 36261 36261 1510	- - 00000358 00000346	- - 34751 (95.8) 1480 (98)	- - 1510 (4.16) 30 (1.99)

The global pointer to vector B can be updated once for each outer loop invocation.
Thus overhead for vector duplications = 30 invocations x 7 cycles/invocation = 210 cycles.

Setup/exit costs = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 2(t_p) + 1(t_s)
- 1 + 1(extra "mov" inst.) = 12 cycles
Total setup/exit costs = 12
Vector length per invocation = 24. No strip-mining of TM necessary.
Execution time: 210(overheads for vector duplication) + 12 x 1510 + 36261 = 54591
CVA-only saving: 506144 - 54591 = 451553

(iii) CVA/PVA executions
Same as CVA-only executions

(iv) CVA Executions Using Various Sizes of TM
No TM:
Total setup/exit costs = 12
2 reads and 1 write are needed to produce 1 result. Not including vector setup, the throughput rate for CVA executions is one result every 2 cycles, due to M0 and/or M1 conflicts.
Execution time: 12 x 1510 + 36261 x 2 = 90642
CVA saving: 506144 - 90642 = 415502

In each invocation, vector length, n = 24 (or 96 bytes).
For TM of size 64 bytes: needs strip-mining
Execution time per invocation: 24 + (n/m+1) x (43 + C_{CVA}) + n = 24 + (24/16+1)x(43+12) + 24 = 158
Total execution time = 158 x 1510 + 390(vector duplications) = 238970 (> 90642). TM not used.

For TM of size 128 bytes or larger: no strip-mining is necessary.
CVA saving = 451373

Table A.13.12: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	Using TM?	Total Exe. Time	CVA-only Cycle Saving	Speedups during executions of loop 5
0	N	90642	415502	5.584
64	N	90642	415502	5.584
128	Y	54591	451553	9.272
256	Y	54591	451553	9.272
512	Y	54591	451553	9.272
1024	Y	54591	451553	9.272

(v) IReq From Core
Base machinne:
IReq from core during loop executions: 10 x 34751 + 9 x 1510 = 361100
PVA-only executions:
IReq from core during PVA executions: 8 (setup code/vector inst.) x 1510 + 3 x 36261 = 120863
CVA-only executions:
IReq from core during CVA executions: 8 (setup code/vector inst.) x 1510 = 12080
CVA/PVA executions: same as CVA-only executions.

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 8 (setup code/vector inst.) x 1510 + 3(essential inst.) = 12083
PVA-only with all inst. caching: same as PVA-only with essential inst. caching.
CVA-only executions: 8 (setup code/vector inst.) x 1510 = 12080
CVA/PVA executions: 8 (setup code/vector inst.) x 1510 = 12080

A.13.6 Summary

Total execution cycles = 2000000
Total cycles in loops = 17019 + 334056 + 550000 + 223169 + 506144 = 1630388 or 81.52%
Average number of iterations per invocation = (50 x 17019 + 50 x 334056 + 7 x 550000
+ 1200 x 223169 + 24.01 x 542405)/1666649 = 173.9

(i) PVA-Only executions

cs-load saving: 170000 (loop3) + 140010(loop5) = 310010
cs-store saving: 4526(loop1) + 74276 (loop4) + 103749(loop5) = 182551
lp-ctl saving: 4495(loop1) + 107880 (loop2) + 160000 (loop3) + 111445 (loop4) + 102239(loop5)
= 486059
Total saving = 310010 + 182551 + 486059 = 978620
Speedup during loop executions =1630388 /(1630388-978620) = 2.501 (perf. imp. = 0.6002)
Overall speedup = 2000000/(2000000-978620) = 1.958 (perf. imp. = 0.4893)

Table A.13.13: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	486059	0.2430
cs-load-oh	310010	0.1550
cs-store-oh	182551	0.0913
Total	978620	0.4893

Total cycles in loops = 17019 + 334056 + 550000 + 223169 + 506144 = 1630388 or 81.52%

(ii) CVA-Only executions

Number of cycles in loops = 17019(loop1) + 223169 (loop4) + 506144(loop5) = 746332
Total saving: 11439(loop1) + 185721(loop4) + 451553(loop5) = 648713
Speedup during loop executions =746332 /(746332-648713) = 7.645 (perf. imp. = 0.8692)
Overall speedup = 2000000/(2000000-648713) = 1.480 (perf. imp. = 0.3244)

(iii) CVA/PVA executions

Total saving: 11439 (CVA for loop1) + 107880 (PVA for loop2) + 330000 (PVA for loop3) +
185721 (PVA or CVA for loop4) + 451553 (CVA for loop5) = 1086593
Speedup during loop executions =1630388/(1630388-1086593) = 2.998 (perf. imp. = 0.6665)
Overall speedup = 2000000/(2000000-1086593) = 2.190 (perf. imp. = 0.5433)

(iv) CVA Executions Using Various Sizes of TM

No TM or TM is 64 bytes (not used).
CVA-only executions:
Total saving: 11439(loop1) + 185721(loop4) + 415502(loop5) = 612662
Speedup during loop executions =746332 /(746332-612662) = 5.583 (perf. imp. = 0.8209)
Overall speedup = 2000000/(2000000-612662) = 1.442 (perf. imp. = 0.3065)

CVA/PVA executions:
Total saving: 11439 (CVA for loop1) + 107880 (PVA for loop2) + 330000 (PVA for loop3) +
185721 (PVA or CVA for loop4) + 415502 (CVA for loop5) = 1050542
Speedup during loop executions =1630388/(1630388-1050542) = 2.812 (perf. imp. = 0.6444)

Overall speedup = 2000000/(2000000-1050542) = 2.106 (perf. imp. = 0.5252)

Table A.13.14: CVA Executions Using Various Sizes of TM

TM Sizes (bytes)	CVA-Only Executions			CVA/PVA Executions		
	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup	Total Cycle Saving	Speedup During Loop Executions	Overall Speedup
0	612662	5.583	1.442	1050542	2.812	2.106
64	612662	5.583	1.442	1050542	2.812	2.106
128	648713	7.645	1.480	1086593	2.998	2.190
256	648713	7.645	1.480	1086593	2.998	2.190
512	648713	7.645	1.480	1086593	2.998	2.190
1024	648713	7.645	1.480	1086593	2.998	2.190

(v) IReq From Core

Base machinne:
IReq from core overall: 1532825
IReq from core during loop executions: 13919+334056+480000+185969+361100 = 1375044

PVA-only executions:
IReq from core during PVA executions:
7936(loop1)+225432(loop2)+190000(loop3)+37386(loop4)+120863(loop5)=581617
IReq from core reduced = 1375044 - 581617 = 793427
Normalized IReq from core during PVA executions = 581617/1375044 = 0.4230
Normalized IReq from core overall = (1532825 - 793427)/1532825 = 0.4824

CVA-only executions:
IReq from core during loop executions: 7936(loop1)+37386(loop4)+120863(loop5)=166185
IReq from core during CVA executions: 651(loop1)+186(loop4)+12080(loop5) = 12917
IReq from core reduced = 166185 - 12917 = 153268
Normalized IReq from core during CVA executions = 12917/166185 = 0.0777
Normalized IReq from core overall = (1532825 - 153268)/1532825 = 0.9000

CVA/PVA executions:
IReq from core during CVA/PVA executions:
651(loop1)+225432(loop2)+190000(loop3)+186(loop4)+12080(loop5) = 428349
IReq from core reduced = 1375044 - 428349 = 946695
Normalized IReq from core during CVA/PVA executions = 428349/1375044 = 0.3115
Normalized IReq from core overall = (1532825 - 946695)/1532825 = 0.3854

Table A.13.15: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.4230	0.0777	0.3115	0.4824	0.9000	0.3854

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during loop executions: 3288(loop1) + 225432(loop2) + 50002(loop3) + 187(loop4) + 12083(loop5) = 290992
Ifetch from memory reduced = 1375044 - 290992 = 1084052
Normalized Ifetch from memory during PVA executions: 290992/1375044 = 0.2116
Normalized Ifetch from memory overall = (1532825 - 1084052)/1532825 = 0.2928

PVA-only with all inst. caching:
Ifetch from memory during loop executions: 3288(loop1) + 2256(loop2) + 50002(loop3) + 187(loop4) + 12083(loop5) = 67816
Ifetch from memory reduced = 1375044 - 67816 = 1307228
Normalized Ifetch from memory during PVA executions: 67816/1375044 = 0.0493
Normalized Ifetch from memory overall = (1532825 - 1307228)/1532825 = 0.1472

CVA-only executions:
Ifetch from memory during loop executions: 651(loop1) + 186(loop4) + 12080(loop5) = 12917
Ifetch from memory reduced = 166185 - 12917 = 153268
Normalized Ifetch from memory during PVA executions: 12917/166185 = 0.0777
Normalized Ifetch from memory overall = (1532825 - 153268)/1532825 = 0.9000

CVA/PVA executions:
Ifetch from memory during loop executions: 651(loop1) + 2256(loop2) + 50002(loop3) + 186(loop4) + 12080(loop5) = 65175
Ifetch from memory reduced = 1375044 - 65175 = 1309869
Normalized Ifetch from memory during PVA executions: 65175/1375044 = 0.0474
Normalized Ifetch from memory overall = (1532825 - 1309869)/1532825 = 0.1455

Table A.13.16: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.2116	0.0493	0.0777	0.0474	0.2928	0.1472	0.9000	0.1455

A.14 Benchmark “ucbqsort”

A.14.1 Critical Loop 1

Table A.14.1: Vertorizing Critical Loop 1

Address	Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000482 00000484 00000486 00000488 0000048a 0000048c	a703 //; ldb r7,(r3) b702 //; stb r7,(r2) 1232 //; mov r2,r3 0593 //; sub r3,r9 0cc3 //; cmphs r3,r12 e7fa //; bt 0x00000482	PVA @L0, @S, #1 cs-store: mov r0, r0[a]	CVA mov @L0,@S;

a. Essential instruction

Table A.14.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000482 0000048c	target bt	675 675	- 00000482	- 378 (56)	- 297 (44)

This loop performs a vector move, C[i] = A[i].
Estimated execution cycles = 9x 378 + 8 x 297 = 5778 or 0.58% of total execution time.
Average number of iterations per invocation = 675/297 = 2.27

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) = 9 cycles
Total setup/exit costs = 9 x 297 + 1(M0 conflict) = 2674
cs-load saving: 3 x 675 - 2674/3 = 1134
cs-store saving: 2 x 675 - 2674/3 = 459
lp-ctl saving: 3 x 378 + 2 x 297 - 2674/3 = 837
Total saving: 1134 + 459 + 837 = 2430
Speedup during loop executions = 5778/(5778 - 2430) = 1.726 (perf. imp. = 0.4206)
Overall speedup = 1017501/(1017501-2430) = 1.002 (perf. imp. = 0.0020)

Table A.14.3: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	837	0.0008
cs-load-oh	1134	0.0011
cs-store-oh	459	0.0005
Total	2430	0.0024

(ii) CVA-Only executions

Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R0) + 1(R2) + 1(exit) + 1(t_p) + 0(t_s) -1 = 9 cycles

Execution time = 9 x 297 + 675 = 3348
CVA saving = 5778 - 3348 = 2430
Speedup during loop executions = 5778/(5778 - 2430) = 1.726 (perf. imp. = 0.4206)
Overall speedup = 1017501/(1017501-2430) = 1.002 (perf. imp. = 0.0020)

(iii) CVA/PVA executions
Total saving: 2430 (using CVA or PVA)
Speedup during loop executions = 5778/(5778 - 2430) = 1.726 (perf. imp. = 0.4206)
Overall speedup = 1017501/(1017501-2430) = 1.002 (perf. imp. = 0.0020)

(iv) CVA Executions Using Various Sizes of TM
N.A. (TM not used).

(v) IReq From Core
Base machinne:
IReq from core overall: 804662
IReq from core during loop executions: 7 x 378 + 6 x 297 = 4428

PVA-only executions:
IReq from core during PVA executions: 7 (setup code/vector inst.) x 297 + 1 x 675 = 2754
IReq from core reduced = 4428 - 2754 = 1674
Normalized IReq from core during PVA executions = 2754/4428 = 0.6220
Normalized IReq from core overall = (804662 - 1674)/804662 = 0.9979

CVA-only executions:
IReq from core during CVA executions: 7 (setup code/vector inst.) x 297 = 2079
IReq from core reduced = 4428 - 2079 = 2349
Normalized IReq from core during CVA executions = 2079/4428 = 0.4695
Normalized IReq from core overall = (804662 - 2349)/804662 = 0.9971

CVA/PVA executions: same as CVA-only executions.

Table A.14.4: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.6220	0.4695	0.4695	0.9979	0.9971	0.9971

(vi) IFetch From Memroy
PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 7(setup code and vector inst.)x297 + 1(essential inst) = 2080
Ifetch from memory reduced = 4428 - 2080 = 2348
Normalized Ifetch from memory during PVA executions: 2080/4428 = 0.4697
Normalized Ifetch from memory overall = (804662 - 2348)/804662 = 0.9971

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only executions:
IReq from core = Ifetch from memory

CVA/PVA executions:

IReq from core = Ifetch from memory

Table A.14.5: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
0.4697	0.4697	0.4695	0.4695	0.9971	0.9971	0.9971	0.9971

A.15 Benchmark “v42bis”

A.15.1 Critical Loop 1

Table A.15.1: Vertorizing Critical Loop 1

Address Opcode //; Assembly Code	Using PVA Construct	Using CVA Construct
00000a0a 950e //; stw r5,(r14) 00000a0c 203e //; addi r14,4 00000a0e 01b6 //; decne r6 00000a10 e7fc //; bt 0x00000a0a	PVA @S, #1 cs-store: mov r5, r5[a]	CVA mov r5, @S;

a. Essential instruction

Table A.15.2: Profile For Critical Loop 1

Address	Entry Type	Execution Counts	Branch Target	Taken count (%)	Not taken count (%)
00000a0a 00000a10	target bt	8271 8271	- 00000a0a	- 8270 (100)	- 1 (0.0121)

This loop performs vector initialization, C[i] = r5, for some scalar r5.
Estimated execution cycles = 6 x 8270 + 5 x 1 = 49625 or 2.48% of total execution time.
Average number of iterations per invocation = 8271

(i) PVA-Only executions

Stall cycles due to M0 conflicts = 1
Setup/exit costs per loop invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit) = 8
Total setup/exit costs = 8 x 1 + 1(M0 conflict) = 9
cs-store saving: 2 x 8271 - 9/2 = 16538
lp-ctl saving: 3 x 8271 + 2 x 1 - 9/2 = 24811
Total saving = 16538 + 24811 = 41349
Speedup during loop executions = 49631/(49631- 41349) = 5.993 (perf. imp. = 0.8331)
Overall speedup = 2000000/(2000000-41349) = 1.0211 (perf. imp. = 0.0201)

Table A.15.3: Performance Improvement for PVA-Only Executions

Types of Overhead Eliminated	Cycle Saving	% Cycle Saving
lp-ctl-oh	24811	0.0124
cs-load-oh	0	0
cs-store-oh	16538	0.0083
Total	41349	0.0207

(ii) CVA-Only executions

Setup/exit costs per invocation = 3(SSR) + 1(CIR) + 2(instr. decode) + 1(R2) + 1(exit)
+ 1(t_p) + 0(t_s) - 1 = 8
Execution time = 8 x 1 + 8271 = 8279
CVA saving = 49625 - 8279 = 41346
Speedup during loop executions = 49631/(49631- 41346) = 5.991 (perf. imp. = 0.8331)
Overall speedup = 2000000/(2000000-41346) = 1.021(perf. imp. = 0.0201)

(iii) CVA/PVA executions

Total saving = 41346 (use CVA or PVA)
Speedup during loop executions = 49631/(49631- 41346) = 5.991
Overall speedup = 2000000/(2000000-41346) = 1.021

(iv) CVA Executions Using Various Sizes of TM

N.A. (TM not used).

(v) IReq From Core

Base machinne:
IReq from core overall: 1660493
IReq from core during loop executions: 5 x 8270 + 4 x 1 = 41354

PVA-only executions:
IReq from core during PVA executions: 6 (setup code/vector inst.) x 1 + 8271 = 8277
IReq from core reduced = 41354 - 8277 = 33077
Normalized IReq from core during PVA executions = 8277/41354 = 0.2001
Normalized IReq from core overall = (1660493 - 33077)/1660493 = 0.9800

CVA-only executions:
IReq from core during CVA executions: 6 (setup code/vector inst.) x 1 = 6
IReq from core reduced = 41354 - 6 = 41348
Normalized IReq from core during CVA executions = 6/41354 = 1.45×10^{-4}
Normalized IReq from core overall = (1660493 - 41348)/1660493 = 0.9751
CVA/PVA executions: same as CVA-only executions.

Table A.15.4: Normalized IReq From Core

During Loop Executions			Overall		
PVA-Only Executions.	CVA-Only Executions	CVA/PVA Executions	PVA-Only Executions	CVA-Only Executions	CVA/PVA Executions
0.2001	1.45×10^{-4}	1.45×10^{-4}	0.9800	0.9751	0.9751

(vi) IFetch From Memroy

PVA-only with essential inst. caching:
Ifetch from memory during PVA executions: 6(setup code and vector inst.)x1 + 1(essential inst) = 7
Ifetch from memory reduced = 41354 - 7 = 41353
Normalized Ifetch from memory during PVA executions: 7/41354 = 1.693×10^{-4}
Normalized Ifetch from memory overall = (1660493 - 41354)/1660493 = 0.9751

PVA-only with all inst. caching: same as PVA-only with essential inst. caching.

CVA-only and CVA/PVA executions:
IReq from core = Ifetch from memory

Table A.15.5: Normalized IFetch From Memory

During Loop Executions				Overall			
PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions	PVA-Only - essential inst. caching	PVA-Only - all inst. caching	CVA-Only Executions	CVA/PVA Executions
1.693×10^{-4}	1.693×10^{-4}	1.45×10^{-4}	1.45×10^{-4}	0.9751	0.9751	0.9751	0.9751

BIBLIOGRAPHY

BIBLIOGRAPHY

[Almasi89] G. Almasi and A. Gottlieb, *Highly Parallel Computing*, The Benjamin/ Cummings Pub. Co. Inc., Redwood City, CA, 1989.

[ALTIVEC98] *AltiVec: The Programming Environments Manual*. Motorola Inc., 1998.

[ARM95] An Introduction to Thumb, Advanced RISC Machines Ltd., March 1995.

[Bajwa97] R. S. Bajwa, N. Schumann, H. Kojima, "Power Analysis Of A 32-Bit RISC Microcontroller Integrated With A 16-bit DSP," *Proc. IEEE International Symposium on Low Power Electronics and Design*, August 1997.

[Bellas98] N. Bellas, I. Hajj and C. Polychronopoulos, "A New Scheme for I-Cache energy reduction in High-Performance Processors," *Proc. IEEE Power Driven Microarchitecture Workshop*, Barcelona, Spain, 1998.

[Bellas99] N. Bellas, I. Hajj, C. Polychronopoulos and G. Stamoulis, "Energy and Performance Improvements in Microprocessor Design Using a Loop Cache," *Proc. International Conference on Computer Design*, Austin, Texas, October 1999.

[Chandrakasan92] A. Chandrakasan, S. Sheng, R. Brodersen, "Low-Power CMOS Digital Design," *IEEE Journal of Solid-State Circuits*, April 1992.

[Chen92] William Chen, Roger Bringmann, Scott Mahlke, Richard Hank and James Sicolo, "An Efficient Architecture for Loop Based Data Preloading," *Proc. IEEE International Symposium on Microarchitecture*, 1992.

[Cray1] Cray Research, Inc., "Cray-1 Computer System Hardware Reference Manual," Bloomington, MN, pub. no. 2240004, 1967.

[CRUSOE00] "Transmeta Breaks the Silence, Unveils Smart Processor to Revolutionize Mobile Internet Computing", company press release, Transmeta Corp., January 19th, 2000.

[Ditzel87] D. Ditzel and H. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," *Proc. IEEE International Symposium on Computer Architecture*, 1987.

[DSP56654] DSP56654 User's Manual, Motorola Incorp., 1999.

[Ghose99] K. Ghose and M. Kamble, "Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation," *Proc. IEEE International Symposium on Low Power Electronics and Design*, San Diego, CA, 1999.

[Horowitz94] M. Morowitz, T. Indermaur and R. Gonzalez, "Low-Power Digital Design," *Proc. IEEE International Symposium on Low Power Electronics*, October, 1994.

[Hwang84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Company, New York, NY, 1984.

[Hwang93] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill Book Company, New York, NY, 1993.

[IA64] IA-64 Application Developer's Architecture Guide, Intel Corporation, 1999.

[IDEN99] "Motorola Introduces i500plus's Multi-Service Digital Wireless Phone - Affordable WML-Compliant Phone with Two-Way Radio, Paging and Internet Capabilities," company press release, Motorola Incorp., August 2nd, 1999.

[Jouppi89] Norman P. Jouppi, Jonathan Bertoni and David W. Wall, "A Unified Vector/Scalar Floating-Point Architecture," Research Report 89/8, Western Research Laboratory, Palo Alto, CA, July 1989.

[Kin97] J. Kin, M. Gupta and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *Proc. IEEE International Symposium on Microarchitecture*, 1997.

[Kiuchi96] Atsushi Kiuchi and Tetsuya Nakagawa, "System with Loop Buffer And Repeat Control Circuit Having Stack For Storing Control Information," *US Patent* number 5,579,493, November 26th, 1996.

[Kissell97] Kissell, *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[Kozuch94] M. Kozuch and A. Wolfe, "Compression of Embedded System Programs," *Proc. IEEE International Conference on Computer Design*, 1994.

[Lee84] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, 1984.

[Lee99a] Lea Hwang Lee, Bill Moyer and John Arends, "Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications With Small Tight Loops," *Proc. IEEE International Symposium on Low Power Electronics and Design*, San Diego, CA, 1999.

[Lee99b] Lea Hwang Lee, Jeff Scott, Bill Moyer and John Arends, "Low-Cost Branch Folding for Embedded Applications with Small Tight Loops," *Proc. IEEE International Symposium on Microarchitecture*, Haifa, Isreal, 1999.

[Lee99c] Lea Hwang Lee, "Low-Cost Embedded Program Loop Caching - Revisited," *University of Michigan Technical Report CSE-TR-411-99*, Department of EECS, University of Michigan, Ann Arbor, December 18th, 1999.

[Lee99d] Lea Hwang Lee, Jeff Scott, Bill Moyer and John Arends, "Data Processor System Having Branch Control and Method Thereof," *European patent* number

EP0965910_A2, December, 22nd, 1999. Patent also pending in US, China, Japan, Taiwan, Korea; filed June 19th 1998, Motorola Incorp.

[Lee99e] Lea Hwang Lee and Bill Moyer, "Data Processing System Having Instruction Folding and Method Thereof," *US patent pending*, filed January 4th, 2000, Motorola Incorp.

[Lee99f] Lea Hwang Lee, "Pseudo-Vector Machine and Method Thereof," *US patent pending*, filed January 11th, 2000, Motorola Incorp.

[Lefurgy97] C. Lefurgy, P. Bird, I. Chen and T. Mudge, "Improving Code Density Using Compression Techniques," *Proc. IEEE International Symposium on Microarchitecture*, December, 1997.

[LEONIA00] "Leonia to Launch Worlds First Secure WebBank Services for WAP Phones", company press release, Leonia Coporation Bank, Finland, January 24th, 2000.

[Lexra99] K. Yarlagadda, "Lexra Adds DSP Extensions," *Microprocessor Report*, vol. 13, no. 11, August 23th, 1999.

[MCORE98] *M•CORE Reference Manual*, Motorola Inc., 1998.

[Martin99] Thomas Martin and Daniel Siewiorek, "The Impact of Battery Capacity and Memory Bandwidth on CPU Speed-Setting: A Case Study," *Proc. IEEE International Symposium on Low Power Electronics and Design*, San Diego, CA, 1999.

[McKee95a] S. Mckee and W. Wulf, "Access Ordering and Memory-Conscious Cache Utilization," *Proc. Symp. on High Performance Computer Architecture*, Raleigh, NC, January 1995.

[McKee95b] S. McKee, "Maximizing Memory Bandwidth for Streamed Computations," *Ph.D. Thesis, Dept. of Computer Sc., University of Virginia*, May 1995.

[Moyer98] Bill Moyer and John Arends, "RISC Gets Small," *Byte Magazine*, vol. 23, No. 2, February 1998.

[Moyer99a] Bill Moyer, Lea Hwang Lee and John Arends, "Data Processing System Having A Cache And Method Thereof," *US Patent* number 5,893,142, April 6th, 1999.

[Moyer99b] Bill Moyer, Lea Hwang Lee and John Arends, "Distributed Tag Cache Memory System And Method For Storing Data In The Same," *US Patent* number 5,920,890, July 6th, 1999.

[NEOPOINT99a] "Neopoint Unveils The Neopoint 1600 Dual Mode Smartphone At PCS'99 In New Orleans", company press release, Neopoint Inc., September 22nd,

1999.

[NEOPOINT99b] “Neopoint Inc. Launches Industry’s Most Comprehensive Location-Based Smart-Wireless Information Service - myAladdin.com”, company press release, Neopoint Inc., September 2nd, 1999.

[PALMVII98] “3Com Announces the Palm VII Connected Organizer, The First Handheld Solution for Out-of-the-box Wireless Internet Access”, company press release, 3Com Corporation, December 2nd, 1998.

[PALMVII99] “Nokia and Palm Computing Ally to Develop a New Pen-based Product Category for Smart Phones”, company press release, 3Com Corporation, October 13th, 1999.

[Patterson96] David Patterson and John Hennessy, *Computer Architecture: A Quantitative Approach*, second edition, Morgan Kaufmann Pub. Inc., San Francisco, California, 1996.

[PowerPC98] *PowerPC 604e RISC Microprocessor User’s Manual*, IBM Microelectronics/Motorola Inc., 1998.

[Scott98] Jeff Scott, Lea Hwang Lee, John Arends and Bill Moyer, “Designing the Low-Power M•CORE Architecture,” *Proc. IEEE Power Driven Microarchitecture Workshop*, Barcelona, Spain, June 1998.

[Scott99a] Jeff Scott, Lea Hwang Lee, Bill Moyer and John Arends, “Assembly-Level Optimizations for the M-CORE M3 Processor Core,” *International Workshop on Compiler and Architecture Support for Embedded Systems*, Washington, DC, October 1999.

[Scott99b] Jeff Scott, Lea Hwang Lee, Ann Chin, John Arends and Bill Moyer, “Designing the M•CORE M3 CPU Architecture,” *Proc. IEEE International Conference on Computer Design*, Austin, Texas, October 1999.

[Segars95] S. Segars, “Panel Discussion: where does the power go?” *International Symposium on Low Power Design*, Dana Point Resort, April 1995.

[Smith88] James E. Smith and S. D. Klinger, “Performance of the Astronautics ZS-Central Processor,” *Astronautics Corporation of America*, March 1988.

[Smith89] James E. Smith, “Dynamic Instruction Scheduling and the Astronautics ZS-1,” *Computer*, vol. 22, no. 7, pp. 21-35, 1989.

[SHARC97] *ADSP-2106x SHARC User’s Manual*, Analog Devices Inc., 1997.

[Smith91] J. E. Smith, “A Study of Branch Prediction Strategies,” *Proc. International Symposium on Computer Architecture*, 1991.

[StarCore98] Ole Wolf and Jeff Bier, "StarCore™ Launches First Architecture," *Microprocessor Report*, vol. 12, no. 14, October 26th, 1998.

[Su95] C. Su and A. M. Despain, "Cache Design Trade-offs for Power and Performance Optimization: A Case Study," *Proc. IEEE International Symposium on Low Power Design*, 1995.

[SuperH98] *SuperH SH-4 Hardware Manual*, Hitachi Inc., 1998.

[Surampudi99] S. Surampudi, "Advanced Battery Systems: Chemistry, Construction and Characteristics," keynote address, *IEEE International Symposium on Low Power Electronics and Design*, San Diego, CA, 1999.

[Thompson98] Scott Thompson, Paul Packan and Mark Bohr, "MOS Scaling: Transistor Challenges for the 21st Century," Intel Technical Journal, third quarter, 1998.

[TMS320C3x] *TMS320C3x User's Guide*, Texas Instruments Inc., 1997.

[TRICORE97] *TriCore Architecture Manual*, Siemens Incorp., 1997.

[Turley98] Jim Turley, "M•CORE M300 Gain Poise, Performance, Second-Generation Motorola Core Enhances Math Ability, Branch Handling," *Microprocessor Report*, vol. 12, no. 16, November 7th, 1998.

[Wolfe92] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proc. IEEE International Symposium on Microarchitecture*, 1992.

[Wulf92] W. Wulf, "Evaluation of the WM Architecture," *IEEE Proc. Int'l Symp. of Computer Architecture*, Gold Coast, Australia, 1992.