ABSTRACT

TECHNOLOGY-ORGANIZATION TRADE-OFFS IN THE ARCHITECTURE OF A HIGH PERFORMANCE PROCESSOR

by

Oyekunle Ayinde Olukotun

Chair: Trevor N. Mudge

The design of computer architectures that reach the performance potential of their implementation technologies requires a thorough empirical understanding of how implementation technologies and organization interact. To help a designer gain this understanding this thesis presents a design methodology that can be used to quantitatively analyze technology-organization interactions and the impact these interactions have on computer performance. All Interactions between technology-organization that affect performance can be analyzed as trade-offs between CPU cycle time (t_{CPU}) and CPU cycles per instruction (CPI), the product of which is time per instruction (TPI). For this reason TPI is used as the performance metric upon which the design methodology is based. Two new analysis tools: 1) a timing analyzer for t_{CPU} and 2) a trace-driven cache simulator for CPI are an integral part of the methodology. The methodology is extensively validated using a high performance processor that is implemented out of gallium arsenide chips and multichip module (MCM) packaging. Besides demonstrating that this methodology is useful for optimizing computer performance, the results of this validation also provide a number of architectural design guidelines.

The task of optimizing the design of the first-level cache of a two- level cache hierarchy is well suited to the methodology because the performance of the first level cache is directly affected by both implementation technology and organization.

To provide the link between implementation technology and organization, simple expressions are derived that provide the access time for on-chip and MCM based caches of various sizes. This access time is on the critical path of the processor and so directly affects t_{CPU}. When the first-level cache has a pipeline depth of one, maximum performance is reached with small caches that have short access times. When the first-level cache is more deeply pipelined, higher performance is reached with larger caches that have longer access times. This is possible because the potential increase in CPI caused by deeper pipelining can be effectively hidden by using static compiler scheduling techniques.

Applying the design methodology to other parts of the cache-hierarchy produces the following results. Second-level caches that are split between instructions and data have performance and implementation benefits. The choice of write-policy (write-back or write-through) depends on the effective access time of the next higher level of the memory hierarchy. Adding concurrency to the cache-hierarchy in the form of write-buffering of the second-level cache or a non-blocking first-level cache provides modest reductions in CPI. When the reduction in CPI is weighed against the increased implementation complexity and possible increase in t_{CPU} these techniques for improving performance are not nearly as effective as increasing the size and depth of pipelining of the first-level cache.

TECHNOLOGY-ORGANIZATION TRADE-OFFS IN THE ARCHITECTURE OF A HIGH PERFORMANCE PROCESSOR

by

Oyekunle Ayinde Olukotun

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:

Professor Trevor N. Mudge, Chairman
Associate Professor Richard B. Brown
Professor Jay W. Chapman
Professor John P. Hayes
Associate Professor Karem A. Sakallah

To my family and friends.

ACKNOWLEDGEMENTS

I would like to thank my dissertation committee members for their careful reading of my thesis. The suggestions that they made have greatly improved the presentation of my ideas. Special thanks go to Trevor Mudge who has been, through out my years of graduate school, an advisor and a friend. Special thanks also go to Karem Sakallah who was instrumental in the development of the timing analysis tools.

I am grateful to my friends in the Robotics Laboratory especially Greg Buzzard, Jarrir Charr, Russell Clapp, Joe Dionse, Paul Gottschalk, and Don Winsor for all the good times we had together.

To the past and present members of the GMIPS group: Ajay Chandra Jeffrey Dykstra, Thomas Hoy, Thomas Huff, Ayman Kayassi, David Nagle, Timothy Stanley, Richard Uhlig, and Michael Upton I thank for making the lab an intellectually stimulating place to be. Special thanks go to David Nagle for providing the Macintosh on which most of this thesis was composed and for taking time to proof read my papers.

Finally, I would like to thank my parents for their love and support
and Liza Barnes for putting up with me for all these years.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

Since 1985 microprocessor based systems have doubled their performance every three years [HP90]. This performance growth, which is better than in any other class of computer, is mainly due to improvements in microprocessor implementation technologies. To sustain this growth rate it will be necessary to use the best implementation technologies available and to optimize microprocessor designs with respect to these technologies. This design optimization process will require a design methodology that uses the low-level constraints of the implementation technology to guide the high-level design decisions of the microprocessor's organization. However, a performance-directed design methodology alone will not be enough to produce optimized microprocessor designs. Analysis tools that can support this methodology effectively will also be required. These tools must be capable of accurately predicting the performance of candidate microprocessor designs so that designers can make quantitative comparisons.

This thesis develops a multilevel optimization design methodology that deliberately takes implementation technology into account when making computer organization decisions. The basis of this design methodology is an analysis of the performance of an architecture using the metric of average time per instruction. Average time per instruction is the product of two components: average number of cycles per instruction and cycle time. To estimate the contribution of each of each of these components to average time per instruction two new analysis tools are used. The tools include a family of trace driven simulators for estimating the average number of cycles per instruction of a particular architecture and a timing analyzer for estimating

the cycle time of a particular architecture. These tools are employed together with the design methodology in the design of a high performance gallium arsenide (GaAs) microprocessor. This design exercise shows that the design methodology can be used to increase computer performance.

1.1 Thesis Overview

The first two chapters of this thesis provide the background and a survey of the pertinent literature, describe the design methodology and present the design tools that are necessary to support it. The last three chapters validate the design methodology by showing how the design of a high performance processor implemented from high-speed technologies can be optimized for performance. Based on this optimization example general conclusions about computer architecture are made.

Chapter 2 presents a measure of computer performance that is based on program execution time. Each of the components of this performance measure is examined and the trade-offs between these components are introduced. These trade-offs are treated in detail in Chapters 4, 5 and 6. Chapter 2 concludes with a review of previous research in the areas of performance evaluation and high performance computer design. This review serves as the departure point for the work presented in the rest of this thesis.

Chapter 3 presents the multilevel optimization design methodology. The methodology explores the computer design space using the new design tools for trace-driven simulation and timing analysis. The motivation for each of these design tools is explained and novel features of the design tools are described in detail. This chapter also introduces the design of a GaAs microprocessor that implements the MIPS instruction set. This microprocessor and the technologies used in its implementation serve as the case study that are used to show the improvement in performance that is possible with the use of multilevel optimization.

Using the multilevel optimization, Chapter 4 examines the problem of finding an optimal primary cache size for a microprocessor. It extends previous work

in this area by accurately considering the effects that varying cache size and degree of cache pipelining have on performance. This chapter shows that if the effects of the implementation technology and organization are considered together the design that results has higher performance than would be predicted by considering either technology or organization alone.

Using the results of Chapter 4, Chapter 5 further applies the design methodology to the design of a two-level cache for a high-performance microprocessor implemented with GaAs and high-density packaging. Again the impact of considering implementation technology on the design of the memory hierarchy are a different and higher performance design.

Chapter 6 presents the conclusions, the limitations and the future directions of the work results presented in Chapters 3–5.

1.2 Contributions

This thesis makes three major contributions:

1. A new design methodology for computer design which includes:
   - the multilevel optimization approach
   - new tools for timing analysis
   - new tools for trace-driven simulation
2. A very extensive validating case study of the design methodology using a GaAs processor.
3. General guidelines for computer design that are the result of the case study.

CHAPTER 2

BACKGROUND

Chapters 3 and 4 present new methods for evaluating the impact of implementation technology in the early stages of computer design. This chapter provides the background necessary to put this work in context. The chapter begins by defining a way of measuring computer hardware performance. This performance metric will be used throughout the thesis to characterize and compare different architectures. Next we asses the impact of VLSI technology on computer performance and argue for the use of GaAs and multi-chip module packaging as high-performance microprocessor implementation technologies. Given a performance metric and an implementation technology a computer designer is faced with the task of detailed performance evaluation to determine the best design. Many performance evaluation techniques have been proposed and some of them are reviewed later in this chapter. Most of these techniques have one of two problems: either they do not predict performance accurately enough to make the right design trade-offs or they neglect the impact of technology entirely. Developing solutions to these problems motivates the multilevel design methodology that will be presented in Chapter 3.

2.1 A Metric for Computer Performance

To make comparisons among computers based on performance and to objectively compare the merits of different design choices we need a way of quantifying computer performance. In this chapter we will define a computer performance metric based on total program execution time that can be used for this purpose. Program

execution time has both hardware and software components. Here we concentrate on the hardware components. Each of the hardware components will be examined and the trade-offs among these components that take place during the course of computer design will be explained. We will also show how these components in turn depend on implementation technology and computer organization factors.

The best metric for computer performance is program execution time [HP90]. However, to make this performance metric an accurate predictor of real world computer performance the programs must be selected so that they are representative of the work environment for which the computer is intended [Smi85a]. More will be said about benchmark selection in Chapter 3.

Program execution time (t_{exec}) can be expressed as

$$t_{exec} = I \times \text{CPI} \times t_{\text{CPU}} \tag{2.1}$$

where I is the number of useful instructions executed, CPI is the average number of cycles per instruction and t_{CPU} is the CPU cycle time. The number I is also referred to as the

instruction path length. The three components of (2.1) depend on four basic factors:

1. implementation technology
2. computer organization
3. instruction set architecture
4. compiler technology

Furthermore, each component of execution time depends on at least two of the basic factors. This interdependence among the components of execution time and the basic factors is illustrated in Table 2.1.

The primary focus of this study is on how the implementation technology and the computer organization factors affect performance. However, as Table 2.1 shows the choice of instruction set architecture (ISA) and compiler affect both the

Factors	I	CPI	T_{CPU}
implementation technology		•	•
computer organization		•	•
instruction set architecture	•	•	
compiler technology	•	•	

Table 2.1: Interdependencies between execution time factors. The dependence of a execution time component on a particular factor is shown by a •.

path length (I) and CPIcomponents of computer performance. In order to isolate the effect that these components have and to make the results of our studies relevant to high-performance computer design in general it is important that we choose an ISA that has been tuned for high- performance implementation.

For the studies in this thesis we have chosen the MIPS R3000 ISA for the following reasons. The MIPS R3000 has good optimizing compilers and analysis tools for generating program address traces. The MIPS R3000 is a reduced instruction set computer (RISC) architecture. RISC ISAs have a number of characteristics that make them well suited to high-performance implementations. Recent studies have shown that, for most programs, RISC ISAs result in longer instruction sequences than complex instruction set computer (CISC) ISAs, such as the VAX, but even with similar hardware implementations, RISC architectures are able to execute the same programs in far fewer cycles [BC91]. Therefore, the execution time of RISC architectures is lower than that of CISC architectures. Furthermore, the simplicity of RISC architectures makes them amenable to implementation in high-speed technologies, where resources are limited [Kat85].

The same instruction set architecture and compiler are used for all performance evaluations, then the instruction path length (I) can be factored out of (2.1), yielding an expression for the average time per instruction (TPI):

$$\text{TPI} = \text{CPI} \times t_{\text{CPU}} \tag{2.2}$$

TPIis a good metric to use to compare different implementations of the same ISA because it focuses the attention on those parts of the performance equation, namely

CPI and cycle time, that are affected by the design of computer hardware. For this reason, the rest of this thesis will use TPI as the metric for evaluating computer performance.

2.2 The CPI-t_{CPU} trade-off

One of the most important trade-offs in computer hardware design is between CPI and t_{CPU}. This trade-off exists because techniques for reducing one of the components of TPI tend to increase the other component. For instance, if a change is made to the computer organization to reduce CPI it is very likely that this change will also increase t_{CPU}. Likewise, changes in organization or technology that are made to reduce t_{CPU} usually increase CPI. This basic trade- off makes it difficult to optimize the performance of a computer by only considering one of the components of TPI. The best performing architecture can only be achieved by looking at the impact of the design choices on TPIas a whole. In order to do this we must gain some insight into the nature of the CPI-t_{CPU} trade-off by looking at how time is spent executing instruction in a computer system. There are two major contributors to the time spent executing instructions in a computer system: first, the time spent executing instructions in the CPU and second, the time spent accessing instructions and data from the memory hierarchy. The next two sections explain how the CPI-t_{CPU} trade-off manifests itself in the design of the CPU and of the memory-hierarchy.

2.2.1 The CPI-t_{CPU} trade-off in the CPU

Typically, RISC architectures use five distinct stages to execute simple instructions, such as ALU instructions, load instructions, store instructions and branch instructions (Figure 2.1).

To maintain a uniform instruction execution process, every instruction must pass through each of these five stages. However, not all instructions make use of all stages. For example, the ALU and branch instructions do not use the MEM stage Figure 2.2 shows the execution time line, of a hypothetical CPU that uses the five

1. IF—instruction fetch
2. RD—read register and instruction decode
3. EX—execution
4. MEM—memory access
5. WB—register write back

Figure 2.1: The five stages of instruction execution.

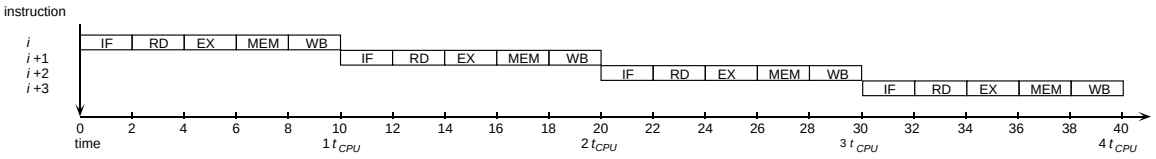

Figure 2.2: Basic CPU instruction execution.

stages shown in Figure 2.1. In this basic CPU a new set of instructions (set size = 1) is issued every 10 time units. If we define t_{CPU} as the shortest time period between two consecutive sets of instructions. Then the t_{CPU} of this CPU is 10 time units. Assuming a perfect memory hierarchy, the CPI for this CPU is 1.0 because a new instruction is issued every cycle. Thus this CPU's TPI is 10 time units. Parallelism and pipelining are two methods for decreasing the TPI of the basic CPU shown in Figure 2.2. Parallelism decreases CPI by duplicating functional unit resources in the CPU so that the instruction issue set size can be increased. Figure 2.3 shows the instruction execution time-line for a machine with an instruction issue set size of two. The CPI of

this CPU is 0.5 while the t_{CPU} remains ten time units. This results in

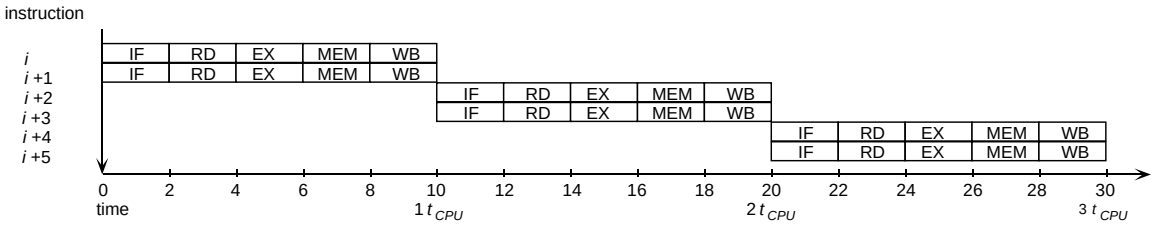

Figure 2.3: Parallel instruction execution.

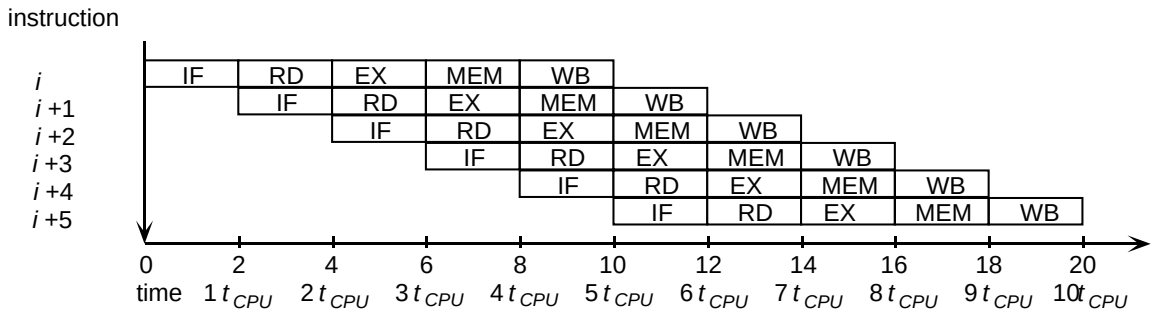

Figure 2.4: Pipelined instruction execution.

a TPI of 5 time units. However, this perfect speedup of a factor of

two over the basic model is achievable providing that two mutually independent (parallel) instructions can be issued every cycle. However, If the instruction stream that is being executed does not contain enough instruction level parallelism to issue two instructions every cycle, the CPI would be greater than 0.5. Furthermore, the implementation of an instruction parallel machine requires the duplication of functional units and register ports. Therefore, given the same implementation technology, the parallel machine will have a longer cycle time due to the greater signal propagation delay and loading of the extra interconnections of the added functional units and register ports. Thus, due to the inherent limits of instruction level parallelism and the increase in cycle time that results from a more complex implementation, there is a limit to the amount of performance that parallel CPU implementations can provide.

In contrast to CPU parallelism, making the CPU pipelined reduces the period of time between the issue of successive sets of instructions (t_{CPU}) by overlapping the stages of instruction execution. Figure 2.4 shows the basic CPU organized as a five-stage pipeline. The t_{CPU} of this CPU is reduced to 2 time units. Assuming a perfect memory system, the CPI is still 1 because an instruction issues every t_{CPU}. This gives a maximum performance of 2 time units per instruction which is equal to the basic machine's TPI(Figure 2.2) divided by the depth of the pipeline, 5. However, there are two factors that prevent most pipelined machines from reaching their maximum performance. First, are pipeline hazards which increase CPI. Pipeline hazards arise when an instruction depends on a previous instruction that, due to pipelining,

has not completed execution. Some pipeline hazards can be eliminated by adding bypassing or forwarding hardware. As an example of a pipeline hazard that cannot be eliminated in this way, consider a load instruction at position i in Fig. 2.4. This instruction will not complete until stage MEM. An ALU instruction following it in position $i+1$ would be forced to stall for one cycle in order to execute in stage EX. Such stalls increase CPI and decrease performance. Second, latch overhead limits the reduction of t_{CPU}. In

order to implement a pipelined CPU, the pipeline stages must be

separated by latches. The delay of these latches causes the pipelined machine's t_{CPU} to be greater than t_{CPU} of the unpipelined machine divided by the pipeline depth. Making a pipeline deeper increases the number of pipeline hazards and makes the latch overhead into a larger fraction of t_{CPU}. This leads to diminishing performance gains as the pipeline is made deeper. Just as in parallelism, there is a limit to the performance increase that pipelining can provide.

2.2.2 The CPI-t_{CPU} in the Memory Hierarchy

This section examines how the CPI-t_{CPU} trade-off manifests itself in

the design of the memory hierarchy. In the following discussion, the focus will be on the cache hierarchy — the part of the memory hierarchy between the CPU registers and main memory. The CPU registers provide the link between the CPU and the cache hierarchy. Instructions and data are fetched from the first level of the cache hierarchy and placed in these registers during the IF and MEM execution stages. Fetching instructions and data may be accomplished in one CPU cycle if the data is present in the first level cache, but if the data is not present in this cache, it can take many CPU cycles to fetch the data. The cache organization determines whether the data will be found in the cache. Table 2.2 presents a summary of the terminology that has been used to describe cache organizations and that we will use in this thesis.

The time it takes for the CPU to access data that hits in the cache is a function of the cache's organization and the cache's implementation technology.

Cache	A small fast memory that duplicates the active parts of a slower, larger memory. A cache is specified by its size, number of sets, associativity, line size, fetch policy, replacement policy and write policy.
Line (Block)	The smallest unit of cache data that is associated with a tag.The tag indicates which line of the main memory is occupying a particular cache line.
Set	A group of two or more cache lines in which a memory line may reside.
Associativity	The cache set size, *i.e.*, the number of cache lines in which a memory line may reside. Caches with a single set of lines are called fully-associative. Caches with a set size of one are called *direct-mapped.*
Fetch policy	The algorithm that determines when and what data to transfer from main memory to the cache. The amount of data that is transferred after a cache miss is called the *fetch size* or *refill size*
Replacement policy	The algorithm that determines which cache line to replace in a set-associative cache. The two most common policies are to replace the least recently used (LRU) line in the set or to replace a line at random.
Write policy	The method that determines what happens when the CPU writes into the cache. The two basic policies are: 1) write-through, in which both the cache and the memory are updated; and 2) write- back, in which only the cache is updated. In the write-back policy, memory is updated when the previously modified (dirty) cache line is replaced.
Miss ratio	The ratio between the number of references that are not found in the cache to the total number for references made to the cache.
Miss or refill penalty	The time it takes to refill the cache after a miss.

Table 2.2: Common cache terminology (adapted from [Smi82])

Typically, the cache's access time grows with its size and set-associativity. In contrast, the cache miss ratio decreases with the cache's size and set associativity. These facts underlie the trade-off between miss ratio and access time. Larger, more set associative caches have lower miss ratios, but longer access times. If the cache determines the cycle time, as the first level cache usually does, this trade-off between miss ratio and access time is equivalent to the CPI-t_{CPU} trade-off. In the higher levels of the cache hierarchy the trade-off between miss ratio and access time only affects CPI.

The trade-off between miss ratio and access time does not completely cover the cache design space. Parallelism (multi-ported) and pipelining can also be applied to improve the performance of caches.

Again, all design decisions can be formulated as trade-offs between CPI and t_{CPU}. The design of cache-hierarchies for high performance microprocessors using these techniques will be covered in Chapters 4 and 5.

2.3 The Impact of Implementation Technology On Performance

A computer's implementation technology consists of the integrated circuits and packaging from which it is fabricated. The performance of this technology is the key factor in determining computer performance. This has always been the case and was recognized in the early days of supercomputer design by Thornton [Tho63]. The performance of a particular combination of technologies can be estimated by the number of gates times the clock frequency that a system implemented in these technologies would have. The resulting estimate is measured in units of gate-Hz and is called computational capacity. The computational capacity determines the speed and organizational complexity of a computer. In general, more complex computer organizations have lower clock frequencies because they require more gates than less complex computer organizations.

To gain some insight into how the increase in computational capacity and

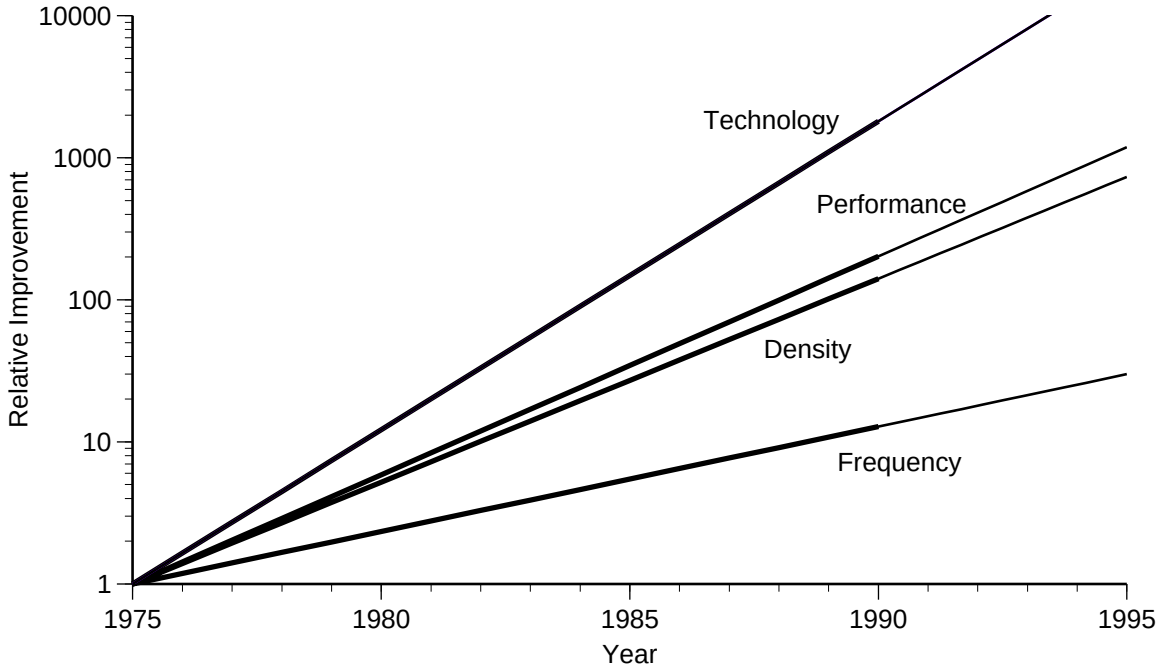

Figure 2.5: Relative improvement of microprocessor performance and VLSI technology [Bak90, HP90, SLL89]

computer performance are related,

Figure 2.5 shows the relative improvement in performance of microcomputers and the relative improvement of VLSI technology over the last fifteen years. The vertical axis shows the relative improvement and the horizontal axis the year. The clock frequency of VLSI designs has increased by a factor of 17 % per year. The integration density (number of logic gates per chip) has increased by 33 % per year. The total improvement from technology is 56 % per year. This assumes that a doubling in clock frequency and a doubling in integration density results in a quadrupling in technology improvement. Furthermore, this rate of increase in technology shows no signs of abating in the near future (next ten years). In the same period, microprocessors have improved in performance (millions of instructions per second) by 35% per year This figure shows that all increases in microprocessor performance can be attributed to either increases in clock frequency, or increases in integration density. In fact, the improvements in technology have outstripped improvements in microprocessor performance. There are three reasons for this. First, the increase in memory speed has not matched the increase in CPU speed. Second, for reasons stated in

Section 2.2.1, adding parallelism to the CPU (more gates) has diminishing returns in performance. Third, microprocessor organizations have not always been optimized to their implementation technologies.

Through the use of the design methodology and computer organization ideas this thesis presents, we will show how microprocessor performance can be optimized for advanced implementation technologies. This methodology is based on an understanding of the interaction between these implementation technologies and computer organization. To provide a basis for the material presented later in the thesis, the rest of this section provides an overview of microprocessor implementation technologies and also introduces a simple model that can used to compare these technologies in terms of computational capacity and power consumption. Simple comparisons like these provide a valuable estimate of the relative capabilities of alternative implementation technologies. These estimates are useful in making design decisions at the organizational level. The section ends with a review of static timing analysis techniques that, given technology parameters, can be used to get more accurate measures of system clock frequency than the simple models.

Microprocessors performance has benefitted most from advances in VLSI technology. VLSI requires transistors with high packing density, low power dissipation and high yield. The metal-oxide semiconductor (MOS) transistor embodies all these attributes, thus making it an ideal candidate for VLSI [Ong84]. Two properties in particular make the MOS transistor a good building block for VLSI circuit designers. First, the MOS gate capacitor does not draw any static current. This serves to isolate each gate from the next stage and makes it possible to build low power circuits with gates that drive many other gates (high fanout). Second, the MOS gate is a capacitor and so it can store charge. This allows a single MOS transistor to act as a memory device, albeit for short periods of time. This capability has been very useful in the design of dynamic logic circuits that have low power and high density [Wan89].

Using CMOS (complementary MOS) transistors it is possible to design and fabricate circuits that dissipate very little power (no static power) and have high noise margins. For these reasons CMOS has become the dominant technology for

both microprocessors and memory. The data in Figure 2.5 show how the integration density and speed of silicon (Si) MOS technology has increased over the last 15 years.

If CMOS technology has a weakness, it is its inability to switch large capacitive loads rapidly. This weakness is becoming more of a liability as die sizes and clock frequencies increase making necessary to switch long, highly capacitive wires quickly. In contrast to CMOS, bipolar devices switch quickly and can drive large capacitive loads, but dissipate large amounts of power. By integrating bipolar devices and CMOS devices on the same die it is possible to have the advantages of both types of device. These BiCMOS (bipolar CMOS) chips use CMOS devices for most of the logic functions and bipolar devices where large current drives or analog circuit functions are required. Careful design of BiCMOS IC's is required to ensure that bipolar devices are used only where necessary in order to avoid excessive power consumption. BiCMOS has been used very successfully in high-speed, high-density SRAMs [Bak90].

Recently, gallium arsenide (GaAs) has emerged as a VLSI implementation technology for high speed applications [LB90]. Gallium arsenide has the advantage of higher electron mobility and higher peak electron velocity at lower electric fields than silicon. This makes it possible to design circuits that switch more quickly over a smaller voltage swing than similar circuits fabricated in silicon. Furthermore, GaAs forms a semi-insulating substrate which reduces the parasitic capacitance of on-chip interconnections. However, GaAs technology has significant drawbacks compared to silicon: GaAs material is more difficult to process and GaAs circuit structures have much lower noise margins. These drawbacks lead to lower chip densities and lower yields [LS88]. Furthermore, because GaAs does not have a stable native oxide, MOS type transistors cannot be fabricated.

The most mature GaAs device is the metal-semiconductor field effect

transistor (MESFET). MESFETs have a Schottky barrier diode as the gate. Unlike the MOS capacitor, this diode conducts static current when the gate is being driven. This limits the fan-out of GaAs circuits and prevents the use of dynamic circuit structures.

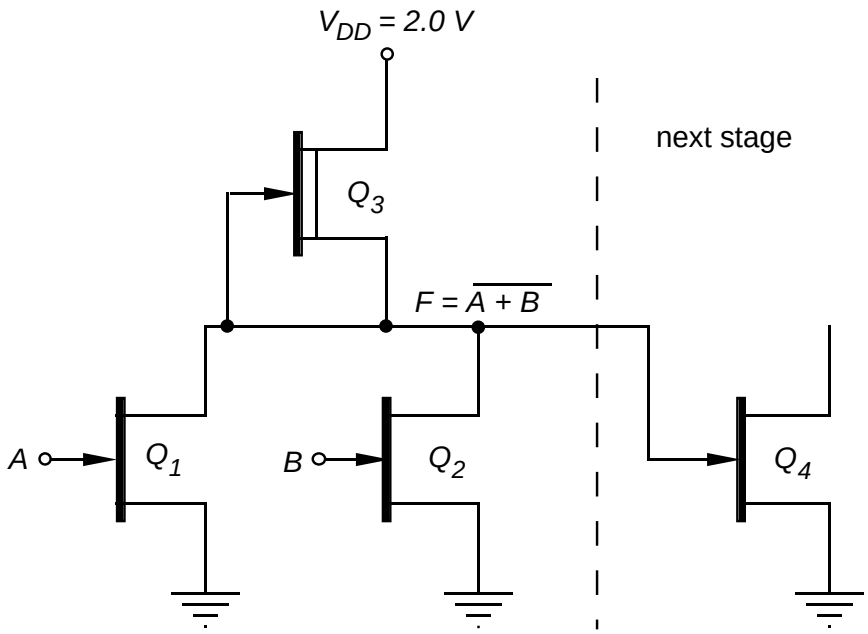

Figure 2.6: Circuit schematic of GaAs DCFL 2-input NOR gate with a fanout of 1. All transistors are E-mode except for Q_3

GaAs MESFETs can be fabricated as depletion mode (D-mode) transistors that are normally on or enhancement mode (E-mode) transistors that are normally off. Figure 2.6 is a schematic of a direct coupled FET logic (DCFL) circuit in which a 2-input NOR gate with a fanout of 1. The structure of DCFL circuits is similar to E/D NMOS circuits except that NAND structures do not work well and pass transistor and dynamic circuits do not work at all. The operation of a DCFL circuit is similar to current steering logic. The Q_3 transistor supplies current that is either steered through Q_1 or Q_2 if either are on, otherwise the current turns Q_4 on. The current steering operation of DCFL has the benefit that it creates very little $\frac{di}{dt}$ noise [HCLC91].

A computer designer would like to know the relative performance of the various VLSI technologies. Although it is impossible to get a completely accurate measure of the performance of a computer design in a particular technology without a complete implementation, it is possible to develop close approximations to performance. Bakoglu has developed a model for CPU performance called SUSPENS (Stanford University System Performance Simulator) [Bak87]. SUSPENS is an an-

alytical model that is primarily used for predicting t_{CPU} based on technological parameters. Organizational parameters that affect t_{CPU} are also included in the model. The factors that SUSPENS models are:

- *Device Properties:* the drive capability and the input capacitance of the transistors.
- *On-chip interconnections*: the number of levels, electrical characteristics, and packing density of on-chip interconnect.
- *Off-chip interconnections*: the number of levels, electrical characteristics, and packing density of the chip-to-chip interconnect.
- *Power dissipation*: the power dissipation density of the module on which the chips are packaged.
- *Implementation*: the average on-chip interconnection lengths, the average number of gates per clock cycle, and the average fan-out per gate.

SUSPENS models interactions between these factors together with Rent's rule [Rad69] to predict the die size, clock frequency ($1/t_{\mathrm{CPU}}$) and power dissipation of the CPU. Rent's rule is an empirical formula that relates the number of I/O pins to the number of gates in a block of logic. This relationship can be expressed as

$$N_p = K N_g^{\beta}$$

where N_p is the number of I/O pins, β is the Rent's constant, N_g is the number of gates in the logic block, and K is a proportionality constant. The constants K and β are determined empirically and change with different CPU organizations (e.g. microprocessors, minicomputers, supercomputers).

Figure 2.7 shows a comparison between CMOS, silicon bipolar and gallium arsenide chips. The CMOS and silicon bipolar data come from

[Bak90]. The GaAs data is based on a new SUSPENS model that we have developed for Vitesse VSC2 technology [VIT91] (see Appendix A). In the comparison,

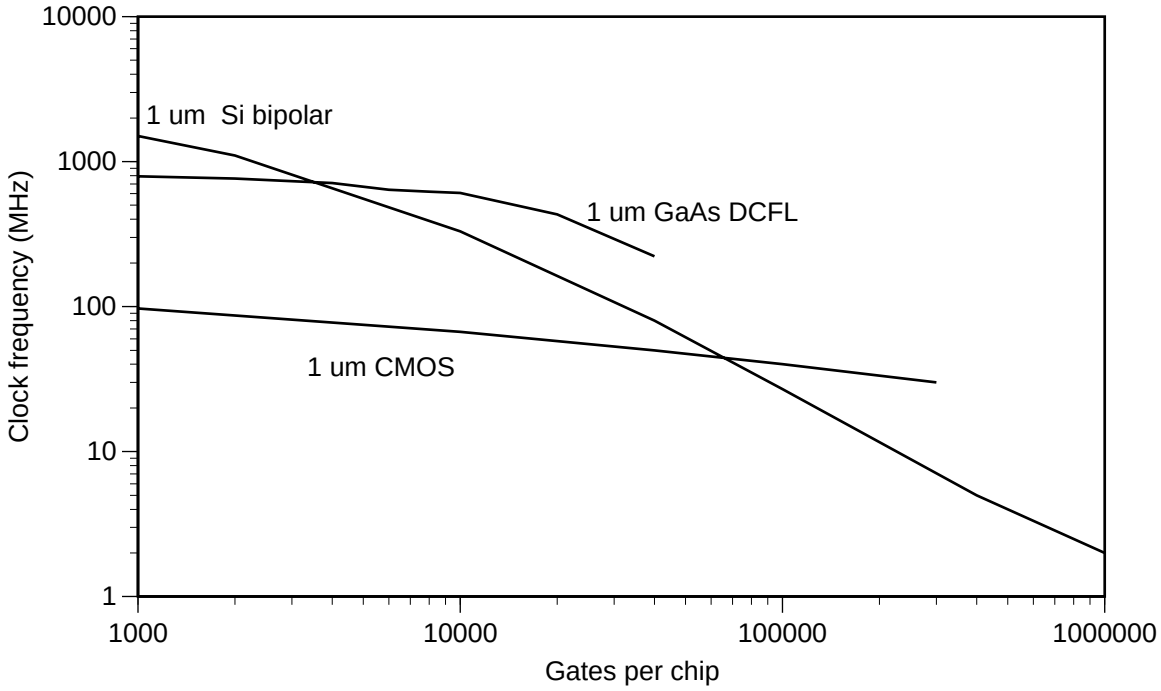

Figure 2.7: A comparison of CMOS, silicon bipolar and GaAs DCFL technologies. The average logic depth is 10 gates and number of interconnection levels is 4 for all technologies.

the transistor length, the number of levels of on-chip interconnect and Rent's constants are kept the same for all three technologies. The data shows that at very low levels of integration (1 000 gates) the silicon bipolar technology has the highest clock frequency. However, GaAs technology has the highest clock frequency for integration densities greater than 4 000 gates.

The power dissipation of each chip is limited to 10 W. This power dissipation constraint severely limits the clock frequency of silicon bipolar chips as the integration density increases beyond 10 000 gates. The power dissipation of a GaAs chip can be reduced by scaling down the average size of the transistors in a gate. At around 40 000 gates the transistors are of minimum size, and no further scaling is possible: it is no longer possible to limit the power dissipation to 10 W for integration levels greater than 40 000 gates. Figure 2.7 shows the line for GaAs ending at 40 000 gates. CMOS reaches the power dissipation limit at 300 000 gates. Figure 2.7 shows the line for CMOS ending at 300 000 gates. The current of bipolar gates can be limited so that the chip always satisfies the power dissipation limit. Doing this slows the bipolar

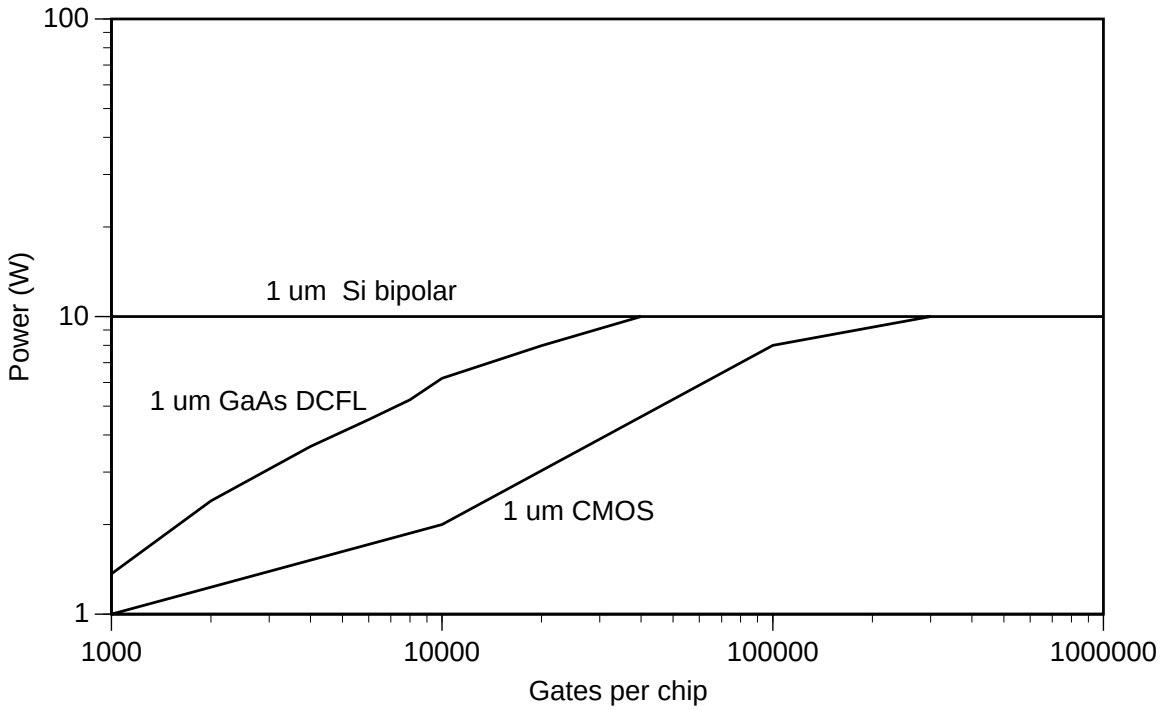

Figure 2.8: Power consumption versus number of gates for CMOS, silicon bipolar and GaAs DCFL.

chips down. Figure 2.8 shows power consumption versus integration density for the three technologies.

The CPU clock frequency of GaAs is determined by the average gate delay and the average number of gates per CPU clock cycle. The gate delay increases linearly with the fan-out of a gate. The fanout of a

gate can be defined as the ratio of the total width of the transistors

that are being driven divided by the width of the driver transistor. In order to include the loading of the interconnect between the driver transistor and the driven transistors the length of interconnect L_{int} can be multiplied by a proportionality constant α_{int} that converts the loading of a unit length of interconnect into equivalent transistor gate width. The fanout (FO) of the gate in Figure 2.6 is given by

$$\mathrm{F0} = \frac{W_4 + \alpha_{int} L_{int}}{W_{driver}} \tag{2.3}$$

where W_4 is the

input gate width of Q_4 of the next stage, L_{int} is the length of the interconnect between the output of the NOR gate and the input of Q_4 and W_{driver} is the gate width

Technology	N_g	D_c (mm)	T_g (ps)	f_c (MHz)	P (W)
CMOS	1 000	1	1 000	97	1
Si Bipolar	1 000	1	65	1500	10
GaAs DCFL	1 000	2	126	792	1
CMOS	10 000	5	1 500	67	2
Si Bipolar	10 000	5	290	330	10
GaAs DCFL	10 000	6	164	607	6
CMOS	100 000	23	2 300	40	8
Si Bipolar	100 000	23	3 50	27	10

Table 2.3: A comparison of CMOS, Si bipolar and GaAs DCFL technologies. N_g is the number of gates, D_c is the length of the side of the square die, T_g is the average gate delay, f_c is the clock frequency and P is the power dissipation of the die.

of driver transistor (Q_1 or Q_2). As integration density increases, the size of the driver is reduced to conserve power, thus reducing its input capacitance. At the same time, the loading of the interconnect increases, because on average the gates being driven are farther away. This results in a drastic increase in fanout as integration density increases. This increase in fanout causes the rapid decline in clock frequency for integration densities from 2 000 to 60 000 gates (see Figure 2.7).

It is possible to compare single chip and multiple chip implementations in the different technologies. Table 2.3, which is based on Figure 2.7, shows that a 100 000 gate CPU implemented as a single chip in 1μm CMOS can achieve a clock frequency of 40 MHz dissipating 8 W of power. A single chip implementation in GaAs DCFL is not possible within the power constraints. A ten chip implementation of the same CPU in GaAs will have a clock frequency of 607 MHz and a power dissipation of 60 W. Assuming that the packaging technology does not affect maximum clock frequency, the power dissipation has been increased by a factor of 7.5, but the performance has increased by almost a factor of 15. Clearly, when the effect of inter-chip interconnection delay is included this performance will be reduced somewhat. At the lowest integration level (1 000 gates) GaAs DCFL achieves half the speed of silicon

bipolar at one tenth the power dissipation.

The integration density and power dissipation of a CPU usually depend on its cost and performance. At the high end of the spectrum are supercomputers, which have low integration density, high clock frequency, high power dissipation and high cost. High performance packaging and cooling are necessary to reduce the impact on system performance of multiple chip crossing delays and to remove the large amounts of heat generated by the chips. The CRAY-1, CRAY-2 and CRAY-3 machines designed by Seymour Cray are the best examples of this class of CPU [Rus78, KH87]. At the low end of the spectrum are single chip microprocessors fabricated from CMOS, with high integration density, low clock frequency, low power dissipation and low cost.

GaAs DCFL technology changes the nature of the trade-offs between integration density, power dissipation and performance. At integration densities of thirty to forty thousand gates, GaAs DCFL technology provides three to four times the performance of CMOS for twice as much power dissipation. In addition, at this integration density GaAs has twice the performance of silicon bipolar for the same power dissipation. For integration densities less than 40 000 gates the current mode logic (CML) used in Si bipolar achieves higher clock frequencies. For integration densities much greater than 40 000 gates the power dissipation of CML becomes unacceptably high. If the power dissipation could be controlled at integration densities greater than 100 000 gates, CMOS technology would have a higher clock frequency. It should be noted that it is possible to implement CML in GaAs and achieve even higher frequencies than Si bipolar, but high power dissipation results and, consequently, clock frequency declines rapidly as integration density increases [Bak90]. Finally, it should be noted that this comparison is based on 1990 technologies. Smaller minimum feature sizes will achieve higher clock frequencies and less power dissipation for the same number of gates. However, the relative speed-power-density product of different implementation technologies is expected to remain constant.

We have seen that at the integration density of forty thousand gates GaAs technology has superior performance and power dissipation to silicon technology. Forty thousand gates is a significant because it provides adequate integration density

to implement a RISC CPU [HCS+87, Jou89]. However, this is an insufficient integration density to implement a cache memory with a miss ratio small enough to make it useful. Therefore it is necessary to use high-performance packaging techniques to ensure the high on-chip performance of GaAs is not wasted by excessive chip crossing delays.

Until recently, high performance packaging had been an area that was

largely ignored by the designers of microprocessor based systems because the speed of the chips used in these systems did not warrant

advanced packaging techniques. However, the speed of the integrated

circuits implemented in high-speed technologies such as BiCMOS and GaAs have reached the point where the interconnections between them now limit total system performance. Furthermore, due to the lower die yield and higher cost of large chips, it could become more cost effective to combine smaller chips that have higher die yield and lower cost with high-performance packaging than it is to use one large chip [SM91]. This situation motivates us to study the impact that high-performance packaging has on the system-level performance of a microprocessor.

High-performance multichip module (MCM) packaging allows a bare chip

(the integrated circuit die without a chip-carrier) to be mounted on a substrate in close proximity to other chips. This arrangement has a

number of advantages over conventional packaging that uses individual chip-carriers and printed circuit boards (PCBs). Mounting the chips directly on the substrate allows many more signal I/Os and power connections to be made. This significantly increases the chip I/O bandwidth. MCM packages also allow much closer spacing between chips than conventional packaging, thus reducing the chip-to-chip propagation delay and increasing the packaging efficiency. The packaging efficiency is measured by the ratio of active circuit area of the chips to area of the package is increased from 10 % for PCB technology to 30–75 % for high density MCMs [Rec89]. MCMs also provide much denser wires than PCBs. These wires have lower values of parasitic capacitance and inductance, but usually have higher resistance per unit length than PCB wires. The combination of

Parameter	ULSI	Thin Film	Ceramic	PCB
No. layers	2–5	4–9	60	40
Wire pitch (μm)	1–4	25–75	90–200	150–250
Typical Size (cm $\times$ cm)	3×3	10×10	10×10	60×60
R_{int} (Ω/cm)	100–1000	3–10	0.4–2	0.5
C_{int} (pF/cm)	1.5	1.0	2.0	0.8
$C_{int}R_{int}$ (ps/cm^2)	150–1500	3–10	0.8- 4	0.4
Dielectric constant	3.9	2.5–4.0	5-10	4.0
Time of flight (ps/cm)	70	50–70	75–100	70
Power Density (W/cm^2)	40	20	4	0.5
Cost	highest	$\longrightarrow$		lowest

Table 2.4: A comparison of on-chip and off-chip packaging Technology [Bak90, Rec89, Tum91].

short wiring distances, high wiring densities and superior electrical properties of the wiring give MCMs low latency and high-bandwidth chip-to-chip interconnections. This combination also makes it possible to use smaller drivers that have lower power dissipation and faster switching times. However, even though the chips on an MCM may dissipate less power, packaging the same number of chips in a much smaller area, greatly increases the power density of an MCM compared to that of a PCB. Therefore, MCM technology must also be accompanied by more efficient cooling technology.

MCM packaging has been implemented in several technologies, including chip on cofired multilayer thick film ceramic on ceramic substrates; and thin film polymers on silicon or ceramic substrates [Tum91]. Table 2.4 compares the wire density, electrical and thermal performance of these technologies with ultra large scale integration (ULSI) and conventional PCB packaging technology. The table shows that the time of flight of electrical signals is roughly equal for all the packaging technologies. However, the large interconnect resistance of the ULSI interconnections causes them to behave as distributed RC lines instead of a transmission lines. The delay of a distributed RC line is proportional to $C_{int}R_{int}$. The interconnections in thin film MCM technology also behave as distributed RC lines. In contrast, the low resistance of the ceramic MCM and PCB interconnections allows transmission line behavior (time of flight) to characterize the delay. The costs in the table show that

ULSI will be the most expensive. The reason for this is that large chips have very low yields which makes them expensive.

Due to the superior electrical properties, the small size and low weight of MCM packages this technology will see wide spread use in the computer industry in the years to come.

2.4 Predicting Computer Performance—A Literature Review

In this section we will review the techniques that other researchers

have used to predict computer performance. Most of these techniques base their predictions solely on CPI. The presentation of these techniques is divided into two sections that discuss the CPU and memory components of computer performance. The discussion within each of the sections covers the spectrum of performance prediction techniques that range from analytical to simulation methods.Because the literature in these areas is extensive, we limit the presentation to those papers that are directly related to the work described in this thesis.

Analytical models provide a low computation time method for estimating computer performance which allows the rapid exploration of large portions of the design space covered by the model. The low-computation time comes at the price of the accuracy of the performance predictions. In contrast to analytical modeling, detailed simulations can provide very accurate performance predictions, but require long computation times. We will see that in practice researchers trade off computation time and prediction accuracy by combining elements of analytical models and elements of simulation.

2.4.1 Predicting CPU Performance

Most researchers have relied on simulation to determine the number of cycles spent processing instructions in the CPU. The most direct simulation technique is instruction interpretation of the program by a simulator that models the CPU

organization. Researchers have used this simulation technique to evaluate a variety of architectures (see for example [SWP86]). The major drawback of this approach is long execution times. It can take from 1 000 to 10 000 real CPU cycles to simulate one model CPU cycle, depending on the complexity of the model. This slow simulation speed severely limits the size of the programs that can be simulated and the parts of the design space that can be explored. To speed up the simulation process and allow the evaluation of larger programs, trace-driven simulation techniques have been developed. The traces consist of a sequence of instruction and data addresses that are generated during the execution of the computer program. These traces can be collected while the program is being simulated. Once a trace has been collected it can be used repeatedly. Trace driven simulation is much more efficient than direct simulation because no explicit instruction interpretation is required in the simulation process. This increase in efficiency can provide an order of magnitude speedup over direct simulation. Trace-driven simulation has been used extensively to study CPU performance [KD78] and memory hierarchies [Smi82].

Recently, very efficient trace-collection techniques have been

developed that are based on direct program execution instead of simulation[MIP88, SCH+91]. These techniques have enabled

the tracing of much larger programs; however, they require a real machine that implements the instruction set architecture of the computer that is being simulated. The idea behind the direct execution technique is to transform the program, by adding extra instructions to the program's object file, so that it monitors its own execution. Such a program transformation must not affect the

normal execution of the program. The monitoring code can be used to generate traces for performance analysis by other programs. Direct execution techniques can generate traces at a rate that is up to two orders of magnitude faster than simulation based trace-collection techniques. In fact, in most cases it is quicker not to save the trace at all, but instead, to perform the CPU or memory hierarchy analysis as the trace is produced. Doing this makes it possible to simulate traces that far exceed the disk capacity of the computer on which the simulation is being performed.

To further increase the speed and generality of performance predictions, researchers have used statistics derived from program traces to construct performance models. These models are then used, instead of the traces, to predict CPU performance. This approach was pioneered by Peuto and Shustek [PS77] who used a CPU timing model to predict the performance of the IBM 370/168 and the Amdahl 470 v/6. A more recent example of this approach is the computer architects workbench [MF88]. This work uses compile time analysis to predict the performance of a variety of architectures. A further example of this approach is the work done by Emma and Davidson [ED87] to develop performance models for CPU pipelines. They present a set of trace reductions to simplify the collection of statistics that characterize the performance of a large class of pipelines. These statistics show that increasing pipeline depth monotonically increases the CPI of a program due to increases in data and control dependencies. However, as we saw in Section 2.2.1, increasing pipeline depth decreases the cycle time (t_{CPU}). They develop an analytical model and use it explore this CPI-t_{CPU} trade-off. The optimal pipeline length is shown to be $n_{opt} = \sqrt{\gamma\alpha}$, where γ is the ratio of circuit delay to latch overhead and α is a factor that is inversely proportional to control and data dependency delay. Other papers have also investigated the CPI-t_{CPU} trade-off for CPU pipelines. The first to do this was a paper by Kunkle and Smith [KS86]. This paper studies the CRAY-1S using simulation of the Lawrence Livermore Loops and analytical timing constraints that build on work done by Fawcett [Faw75]. Their results show that 8 to 10 levels of gate delay per pipe segment yields the optimum overall performance for the first 14 Lawrence Livermore Loops [McM72]. Dubey and Flynn have developed an analytical model to explain these results [DF90]. Their equation for optimal pipeline depth has the same flavor as that of Emma and Davidson's.

2.4.2 Cache Hierarchy Performance

There are four aspects to cache design: the cache organization, the performance evaluation methodology, the performance metric and the implementation technology. The goal of cache design is to determine a cache organization that max-

imizes performance given certain implementation technology parameters such as circuit speed, cost, physical size and power consumption. However, the choice of performance evaluation methodology and performance metric will affect this design choice. The dominant method of cache performance

evaluation is trace-driven simulation [Smi82]. In this review of cache design we will show how researchers have evolved from evaluating caches with short single-application traces using miss ratio as the performance metric to evaluating caches with long multiprogramming traces using execution time as the performance metric. This evolution has deepened the understanding of cache behavior and has resulted in improved cache designs. This review will also show how researchers have incorporated technology parameters into the cache design problem. Finally, we will review techniques for speeding up cache performance evaluation through the use of novel trace-driven simulation techniques.

A. J. Smith has written a comprehensive summary of cache design that

analyzes most of the cache organization design alternatives listed in Table 2.2 with trace-driven simulation using miss ratio as the performance metric [Smi82]. Smith gives flexibility and repeatability as reasons for using trace-driven simulation, but also cautions that cache performance is very dependent on the workload [Smi85a]. A real workload includes the effects of multiprogramming and operating system (OS) code which are difficult to simulate with the direct simulation trace generation method used by Smith. To simulate the effect of multiprogramming, Smith creates a multiprogramming trace by interleaving the traces from 3 or 4 programs together using a round-robin scheduling policy with 10 000 memory references between context switches. The total length of the multiprogramming traces was one million memory references. In a later paper, Smith argues that the caches should be flushed between context switches to reduce the sensitivity of the results to the number of programs used to simulate multiprogramming [Smi87].

Measurements taken from real computer systems showed miss ratios that

were higher than those predicted by trace driven simulation [Cla83]. This observation has led researchers to obtain longer traces that include realistic OS and

multiprogramming memory references [ASH86, AKCB86]. In particular, Agarwal's results show that: 1) OS code can have a miss ratio that is twice as high as application code; 2) purging the cache on context switches severely increases the miss ratio of large caches; 3) context switch intervals are not constant; and 4) trace lengths for multiprogramming traces should be longer than one million memory references [AHH88].

The trend in cache research to use more realistic traces was accompanied by a change in the performance metric from cache miss ratio to effective memory access time [Smi87, Aga88]. Effective memory access time (t_{eff}) is defined as $t_{eff} = t_{cache} + M_{cache} \times t_{miss}$, where t_{cache} is the access time of the cache, M_{cache} is the miss ratio of the cache and t_{miss} is the time it takes to fetch a cache line from main memory. Effective memory access time is a better performance metric for a computer with a cache than miss ratio because it reflects execution time. The use of effective access time directly exposes the CPI-t_{CPU} trade-off for caches. In contrast, the use of miss ratio is only indirectly related to this trade-off. As an example, consider organizational changes such as increasing set-associativity or line size that decrease miss ratio, but that may increase t_{cache} or t_{miss}. The result of such changes will be a lower miss ratio but may be a higher effective access time.

The values of t_{cache} and t_{miss} can only be determined with

knowledge of the implementation technologies of the computer. This knowledge is vital, as Hill shows in [Hil87], in order to arrive at a cache organization that maximizes performance. Using effective access time, Hill compares the miss ratios of caches of varying set associativities. The absolute miss ratio difference between set-associativities decreases as caches get larger. Furthermore, for caches of 32 KB or larger the miss ratio reduction

from set-associativity does not make up for the increase in t_{cache} and results in a larger effective access time. Hill estimates the increase in t_{cache} from set-associativity for caches implemented in TTL, ECL and custom CMOS technologies.

Przybylski has used Agarawal's improved traces to evaluate various cache organizations [Prz90a]. He makes a cogent argument for using execution time instead

of miss ratio as the cache performance metric. This metric allows the exploration of trade-offs among cache size, cache cycle time, set associativity, cache cycle time, cache line size, and main memory speed. The trace-driven simulator used to evaluate these trade-offs keeps an accurate account of all cycles spent in the memory system. Przybylski's results show that for common implementation technologies, performance is maximized for cache sizes in the 32–128 KB range. His set- associativity results are similar to those of Hill. He shows that the cache line size that maximizes performance is much smaller than the line size that minimizes the miss ratio [Prz90b]. One of the major conclusions of Przybylski's study of single-level caches is that for high performance processors, multi-level cache hierarchies are needed in order to bridge the gap between CPU cycle time and main memory access time.

On-chip microprocessor caches are of special interest because they must be implemented in a limited amount of silicon area [SH87]. Alpert and Flynn have analyzed on-chip caches of equal area [AF88]. Their results show that when overhead of tag area is included in the silicon area that is consumed by the cache, large block sizes give the highest performance. Furthermore, when the overhead cost is high, caches that fetch partial blocks perform better than caches that fetch entire blocks. Hill and Smith have also investigated on-chip microprocessor caches and have come to similar conclusions [HS84]. Duncombe has investigated how the access time of on-chip caches varies with caches-size and aspect ratio [Dun86]. He shows that the aspect ratio of a cache can have a significant effect on access time. For this reason cache organizations that could be implemented as square arrays had the fastest access times for their size.

Recently, many researchers have investigated two level caches [BW87, BW88, BWL89, BKB90, BKW90, PHH89, SL88, Smi85b, Wil87]. Two-level caches have been motivated by the need for a fast uniprocessor to close the speed gap between the CPU and memory [RTL90] and the need for shared-memory multiprocessors to reduce bus-traffic [LLG+90, LT88]. One of the first comprehensive studies of the performance of two-level caches was by Short and Levy [SL88]. They evaluated a variety of two-level cache organizations based on execution time using a trace-driven cycle-by-cycle

simulator. They concluded that second-level caches provide performance gains when main-memory access times are long or when first-level caches are small. Short and Levy also investigated the effect of varying the write policy at both levels of cache, but did not consider write-buffering. Przybylski extends their work by examining the trade-off between miss ratio and second-level cache access time [PHH89]. His results show that the first-level cache reduces the number of references to the second-level cache, without reducing the number of second-level misses. This has the effect of skewing the trade-off in favor of the miss ratio over the access time, because the second-level cache is not referenced as much as it would be in a single level system. Przybylski concludes that for the best performance the second-level

> caches should be larger and more set-associative, to lower the miss ratio, than a single-level cache in the same system would be. All experiments were conducted with a write-back first-level cache. The write-policy of the second-level cache was not considered.

Besides the techniques for efficient trace generation that were described in the section on CPU performance, researchers have exploited the properties of the traces themselves to reduce the trace-length and thus speedup cache simulation. Smith was the first to develop this approach in a technique called *trace deletion* [Smi77]. This technique, which is based on page reference traces, deletes references to the top D elements of an LRU stack.

Smith shows that the reduced trace can be used to simulate a memory larger than D pages to produce the same number of page faults, providing that the same page size and the LRU replacement algorithm are used. Puzak has extended Smith's technique to the simulation of set-associative caches [Puz85]. Puzak uses a reduced trace generated by references that miss in a direct-mapped cache (cache filter) to simulate caches which have a greater degree of set associativity or larger size. Other techniques for trace compression can reduce the trace significantly, however, they cannot guarantee accurate performance evaluation [Aga88, LPI86], especially

> when the memory-system uses concurrency, *e.g.* overlap between instruction and data references or write buffers.

Cache simulation speed can also be increased by efficiently simulating multiple cache configurations at the same time. Mattson *et al.* have determined the miss ratios of all caches with a certain block size with one pass through the trace [MGS+70]. Their simulation approach is based on the fact that stack replacement

algorithms like LRU guarantee the *inclusion* property. The inclusion property holds whenever the contents of a smaller cache are always a subset of the contents of a larger cache. When inclusion

holds it guarantees that a reference that is a hit in a smaller cache

will always be a hit in the larger cache. Mattson's stack simulation

technique has been extended to cover set-associative caches and various block sizes [TS71, HS89]. Stack simulation has also been adapted to the simulation of write-back caches, where the number of dirty-misses is an important parameter [TS89]. However, even though stack simulation techniques can be used to speed up simulation by as much as an order of magnitude, they cannot be used to predict the performance of cache-hierarchies with prefetch, or write-back buffers, or that use concurrency to improve performance.

sectionThen Why This Research?

Microprocessor implementation technology has progressed to the point

where it is now possible in CMOS to put the CPU, floating point unit

(FPU), cache management unit (CMU), and the first level of cache on

single chip. Using advanced packaging techniques with high speed technologies that have less integration density than CMOS, such as GaAs, it is possible to put the CPU, other system level components and an appreciable amount of cache on a single substrate. In order to

fully exploit the performance potential of such systems it is necessary that the system organization be optimized to the technology. Although researchers have made significant contributions to computer design and computer performance evaluation, except for a few notable exceptions [KS86, Hil87], most have ignored the quantitative exploration of the interaction of technology and organization. Research in this area is particularly important if microprocessor performance is to keep pace

with technological improvement.

This thesis will use a multilevel optimization design methodology to quantitatively explore the effect of technological constraints such as gate-speed, integration density, interconnection delay and system partitioning, together with organizational variables such as parallelism, pipelining and memory hierarchy design on performance as measured by average time per instruction (TPI). Putting these disparate issues in terms of TPI clarifies the trade-offs between them and makes it possible to optimize computer design in a straightforward way.

> Our focus will be limited to the memory hierarchy, because the memory hierarchy determines, to a large extent, the performance of a computer, particularly high performance computers. Furthermore, the memory system components use the majority of the system resources and dissipates most of the power.

CHAPTER 3

A MICROPROCESSOR DESIGN METHODOLOGY

The goal of this chapter is to introduce a new performance directed design methodology for microprocessor design. This methodology will be used in later chapters to optimize the design of a GaAs microprocessor. The base architecture of this processor will also be introduced in this chapter. The implementation of this design methodology requires performance analysis techniques that are capable of accurately predicting TPI. The development of two new performance analysis tools for this purpose will be explained and evaluated using the GaAs microprocessor architecture.

3.1 Multilevel Optimization

Many design automation researchers have advocated a top-down methodology for digital system design. In this methodology, the design starts with an initial high-level specification of the organization which is decomposed and refined into a low-level physical implementation. In the process, the design is transformed between different abstraction levels (*e.g.* register transfer level (RTL), gate level, transistor level), and is optimized separately at each level. Top-down design makes it possible for a team of designers to manage the immense complexity of present day digital systems design; however, because interaction between the abstraction layers is ignored, a purely top-down approach will lead to computer designs with a performance that is lower than the capabilities of the implementation technology.

It has been recognized for some time that the low-level physical characteristics of a particular technology should be used to guide higher level design decisions. Mead and Conway, in their classic book on VLSI design [MC80], stress that physical design issues, such as layout, interconnect and timing, are critical to the quality of the design, and therefore, should be considered early in the course of a design. More recently, researchers in the area of high- level synthesis of digital systems from abstract behavioral level specifications have also recognized the importance of physical implementation issues on the performance of the design [MK90]. In both of these cases the metric of performance has been cycle time.

In the previous chapter it was argued that TPI is the correct performance metric to use to evaluate computer hardware. Given TPI as the performance metric, it is not enough just to consider how implementation technology considerations should guide organizational decisions to achieve a lower cycle time (t_{CPU}), it is also necessary to consider how these design decisions affect the average number of cycles executed per instruction (CPI). To use TPIto guide the design requires an integrated design methodology. Our integrated design methodology is called multilevel optimization. In multilevel optimization, optimizations are performed across traditionally separate abstraction levels to achieve higher system performance. Multilevel optimization relies heavily on the ability to accurately predict the performance at each abstraction level in order to evaluate the impact of design trade-offs on system performance, and may require design iteration among the levels as the design progresses.

The performance prediction tools that are necessary to support multilevel optimization are a trace driven simulator and a timing analyzer. Trace driven simulation provides a an accurate prediction of CPI and timing analysis based on implementation technology dependent delay parameters is used to accurately predict t_{CPU}. During delay analysis, close attention must be paid to the delay caused by the interconnect and multiplexing of a design to ensure accurate delay predictions. Accurate delay predictions are important because, when implementation details are considered, the design space can be discontinuous [MK90]. In a discontinuous design space, a small change in organization or in a particular delay may cause a significant

decrease or increase in t_{CPU}.

A base architecture is defined as a register transfer level (RTL) organization and a set of implementation technologies. It is important that the base architecture be as simple as possible in order to accurately assess the performance impact of making it more complicated. Taking the base architecture as a starting point in the design space, and using the performance analysis tools, multilevel optimization proceeds as follows. The base architecture is simulated to establish a base level of performance. In order to explore the effect of changes in the base architecture a set of candidate designs is encoded into the trace-driven simulator and changes to the RTL model are made to reflect the new organization. The performance of the set of designs is simulated using trace-driven simulation to reveal the general nature of the CPI-t_{CPU} trade-off. Once this trade-off has been established, delay modeling, timing analysis and layout provide the technology constraints. These technology constraints make the CPI-t_{CPU} trade-off specific and allow the selection of the highest performing design choice. The RTL specification represented by this design choice becomes the new base architecture. This process of generation and evaluation of organizations continues until the design meets the performance specifications. The resulting multilevel design methodology is diagrammed in Figure 3.1.

This thesis presents the multilevel optimization of a high performance microprocessor implemented in GaAs and multichip module technology. Through the analysis of this specific example we will draw some general conclusions about the interaction between the organization and implementation of microprocessors. The next section describes a base architecture. This is followed by sections on the performance prediction tools. In the chapters that follow, this architecture will be improved using multilevel optimization.

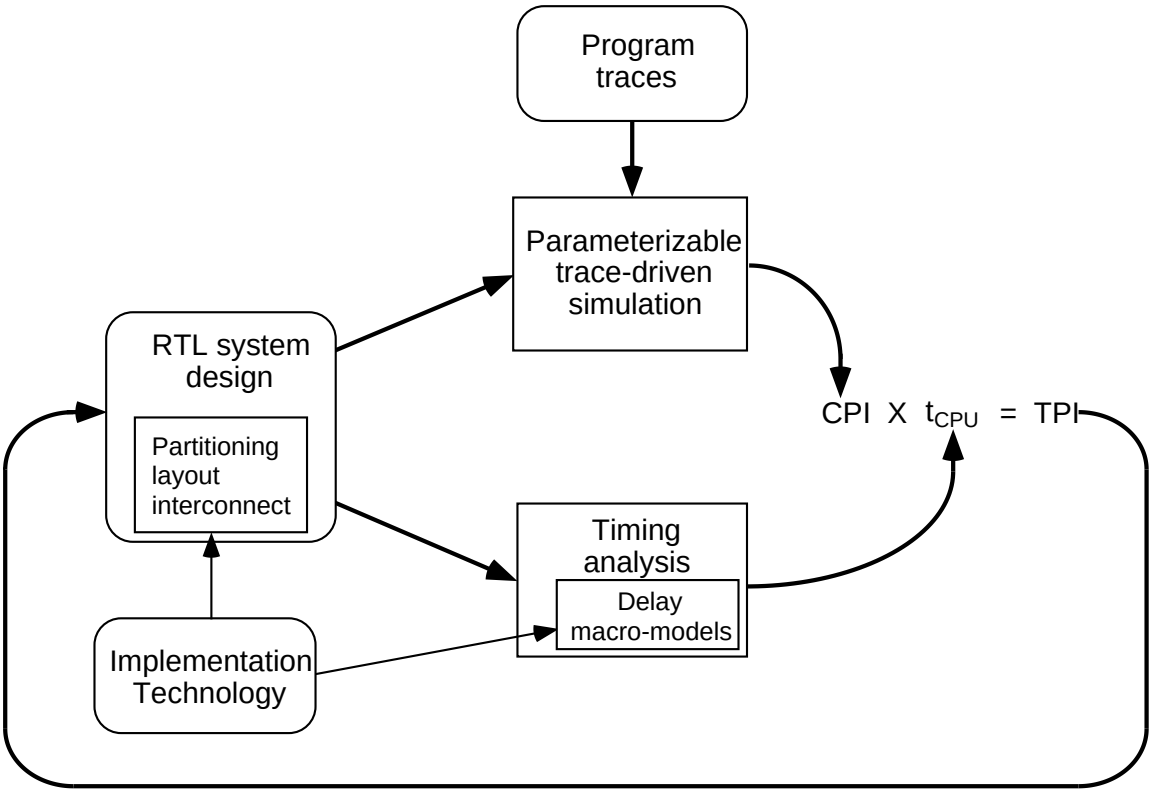

Figure 3.1: Multilevel design optimization.

3.2 Base Architecture

3.2.1 System Level Organization

The system level organization base architecture that serves as the starting point for the design example and that is used throughout this thesis is diagrammed in Figure 3.2.

This architecture includes a single-chip pipelined integer CPU and a single-chip floating-point processor unit (FPU) both of which are based on the MIPS instruction set architecture [Kan87]. The architecture also includes a two-level cache which comprises of a high-speed, low-latency primary (L1) cache and a much larger, high bandwidth secondary (L2) cache. The L1 cache, which is split into I-cache (instructions) and D-cache (data), and the L2 tags are to be implemented with 1K $\times$ 32-bit SRAMs. Finally, the base architecture includes a cache management unit (CMU) chip. The CPU, FPU, CMU and SRAM chips are being designed and fabricated in

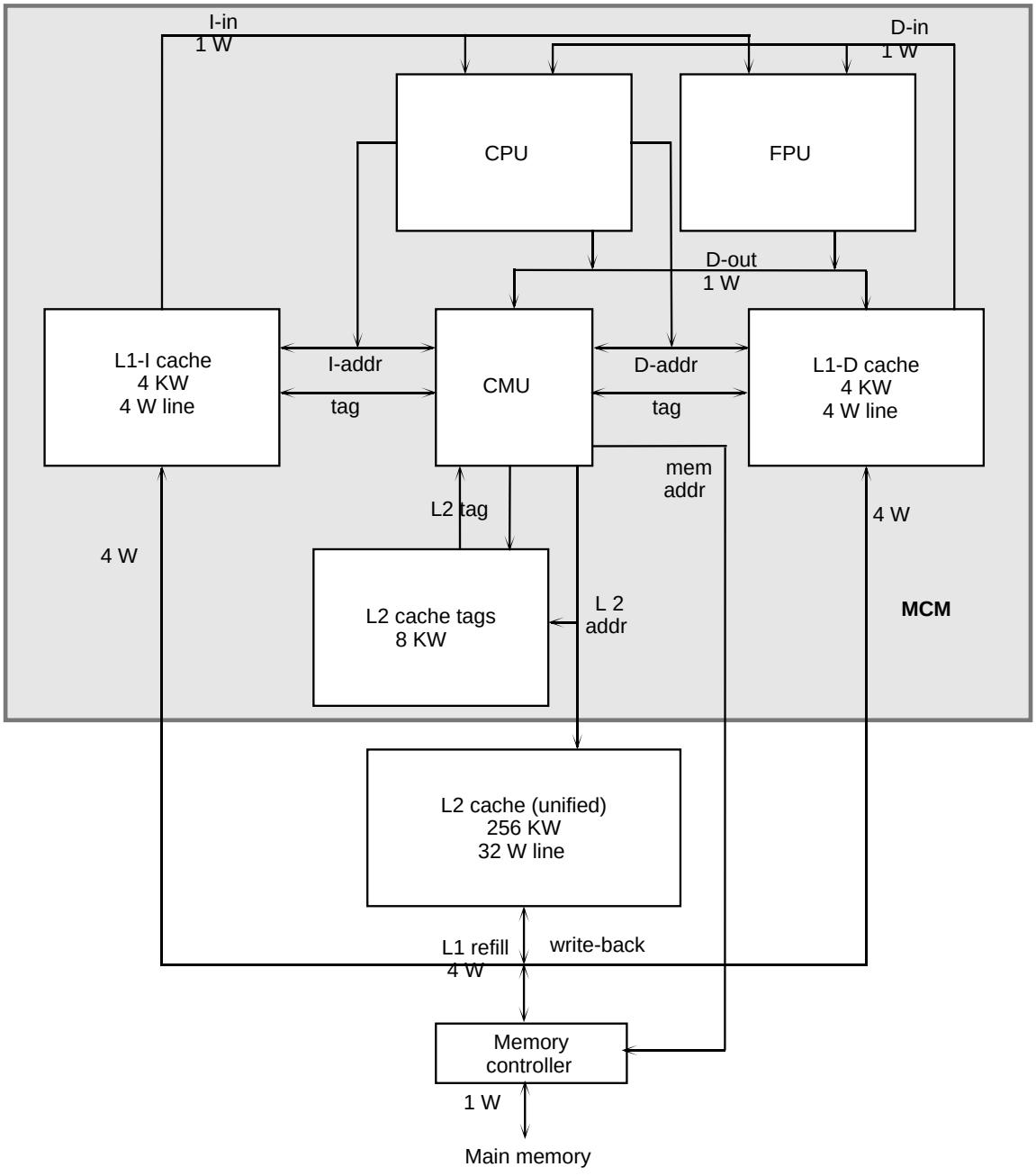

Figure 3.2: The base architecture.

Vitesse's HGaAs III process [VIT89]. All of these chips will be mounted as unpackaged die and interconnected on a MCM. The boundaries of the MCM are shown in Figure 3.2 by the grey region. The target cycle time for this architecture is 4 ns. This will provide a peak instruction execution rate of 250 million instructions per second (MIPS).

The rationale behind the base-architecture is to design an organization that is a simple as possible using GaAs and MCM technology. We will see that as the thesis progresses the design will become more complex. However, complexity is added only if it provides a measurable impact in performance and is feasible within the constraints of the implementation technology. This is the essence of the multilevel optimization design methodology.

In order to provide enough memory for a processor of this speed, assuming several megabytes are needed per MIP, main memory sizes in the range of 512 MB to 4 GB are required[Jou90]. Main memories of this size require hundreds of DRAMS and have access times of between 300 and 400 ns. To hide this main memory access time from the CPU requires the use of a two-level cache hierarchy. The details of this cache-hierarchy are presented below.

The L1 cache provides single cycle read access to the CPU. It is organized as separate direct-mapped 4 KW[1] instruction (L1-I) and data (L1-D) caches. The line size and fetch size are both 4 W. A direct mapped organization is chosen for the L1 cache because this provides the fastest effective access time [Hil88]. The reason for selecting a 4 KW cache size is that the OS page-size is 4 KW. Thus it is possible to use untranslated bits to index directly into the cache without the need for extra hardware resources to solve the synonym problem [WBL89]. Once a 4 KW cache size has been selected, 4 W is the largest refill and line size that can be used to refill a cache line in a single access, assuming the cache is constructed from 1K $\times$ 32 b chips. Single access refill greatly simplifies the L1 cache-control logic. The penalty for refilling a 4 W line from the L2 cache is 6 CPU cycles.

[1] 1 W = 4 B = 32 b; one word = four bytes = 32 bits

The behavior of writes (stores) to the L1-D cache follows a simple write-through write-miss-invalidate policy. Writes that hit in the cache take a single cycle. In this cycle the cache tags are checked, and the data are written into the cache. If the address hits in the cache, then CPU processing continues. If the address misses in the cache, then the CPU must stall for a cycle while the line is invalidated, since it now contains corrupted data. No data are lost because all data that is written is sent to an 8-entry deep write-buffer that is placed between L1-D and L2. This write-buffer allows the CPU to continue processing using the single cycle access L1 cache while the data is written to the slower L2 cache. When an L1 cache miss occurs the CPU has to stall until the write buffer has been emptied and the cache line has been refilled. Waiting for the buffer to empty before fetching the new line from the L2 cache is the simplest way to ensure that ordering of reads and writes in the instruction stream is preserved with respect to the L2 cache.

Details of the L2 cache organization are as follows. Following the lead of earlier work on two-level caches, we use a unified direct-mapped L2 write-back cache [SL88, WBL89, Prz90a, TDF90]. The size of this cache is 256 KW and it is constructed from 8K $\times$ 8 b BiCMOS SRAMs that have an access time of 10 ns. A cache of this size will not fit on the MCM, and therefore it must be placed on a PCB. The extra communication delay from the MCM module to the board adds 2 cycles to the 4 cycle L2 cache access time, yielding the refill penalty of 6 cycles. The refill path between the L2 and L1 caches is 4 W wide and is shared between the L1-I and L1-D caches. To compensate for the large main memory access time the line size and refill size of the L2 cache is 32 W. The actual refill penalties for this line size are roughly 140 CPU cycles if the line to be replaced has not been modified (clean) and almost 240 cycles if the line has been modified (dirty). These penalties assume a main memory bus with a peak bandwidth of 266 MB/s [Tho90]

The CMU contains the translation lookaside buffer (TLB) logic, the write buffer, and the cache controllers for both levels of cache. The TLB is organized as a 2-way set-associative 32 entry cache for instructions and a 2-way set-associative 64 entry cache for data. It requires 4 Kb of memory. The TLB is managed by software

[Kan87]. The write buffer is a 64 b wide 8 entry shift register. The cache controller manages the refill of the L1 cache from the L2 cache and the refill of the L2 cache from main memory. The cache controller also manages the emptying of the write buffer into the L2 cache. Placing the L2 cache tags on the MCM reduces their access time. This is important because the L2 is a write-back cache and so the cache tags must be checked before the write is performed. If the cache tags were constructed from the same SRAMs as the L2 cache and placed on the PCB the L2 write time would double.

3.2.2 Register Transfer Level Organization

An RTL schematic of the CPU and L1 cache datapath used in the base architecture is shown in Figure 3.3. The major components of the datapath are the following combinational circuits:

- *Register File:* The 32 W $\times$ 32 b register file (RF), with two read ports and one write-port. The RF is capable of performing a read or write in half of a CPU cycle.

- *ALU:* The arithmetic and logic unit that is capable of producing a new result every cycle.

- *Shifter:* The barrel shifter, which is capable of left or right shifts of 0 through 31 bits in a single cycle.

- *Comparator:* The comparison unit which is capable of comparing two operands for equality in under half of a cycle.

- *Next-PC:* The next program counter (PC) unit, which contains a PC incrementer for computing the PC for the sequential instruction, and a PC adder for computing the branch target.

- *L1-I cache:* The data half of the level-one cache, which has a single cycle access time.

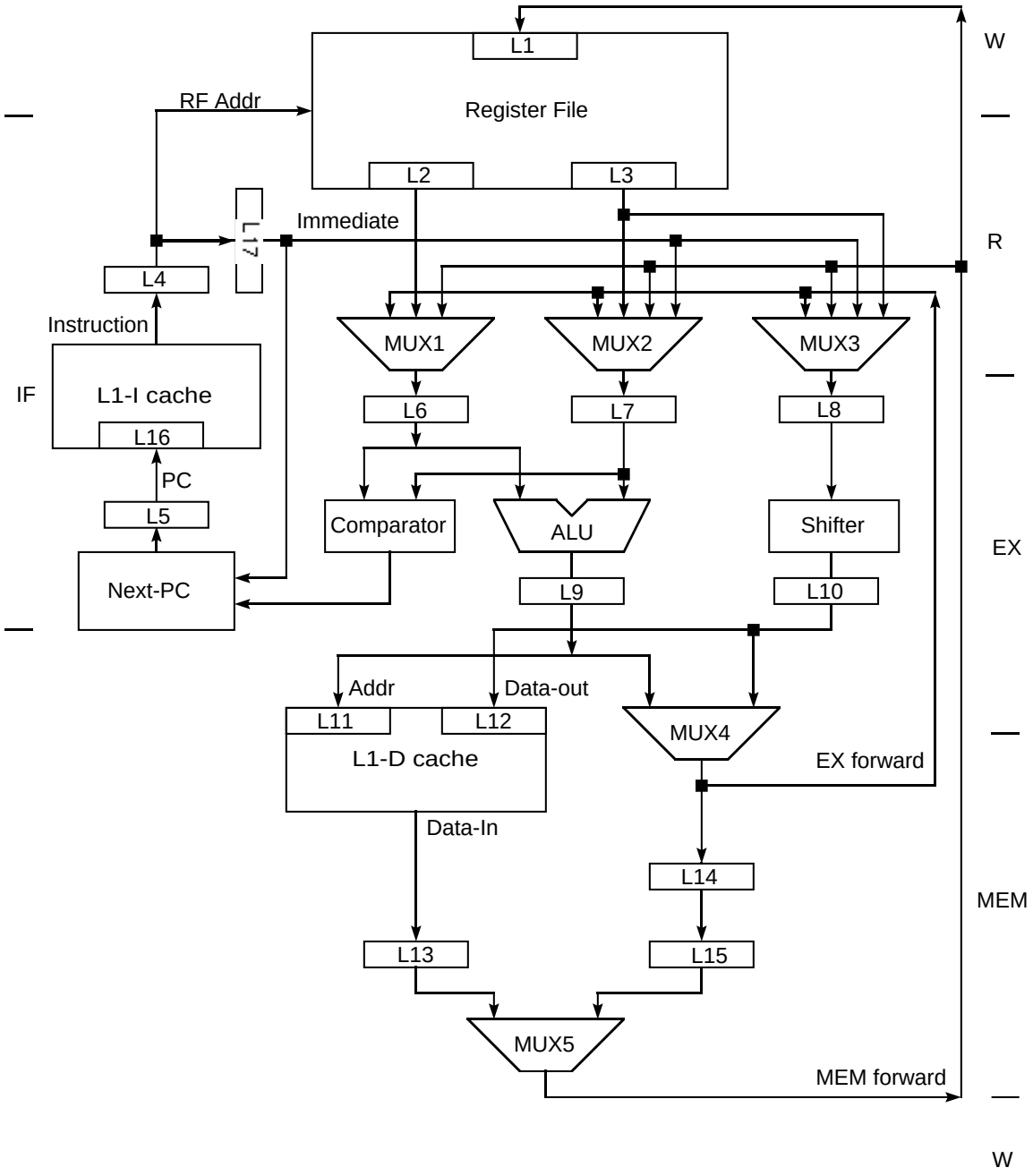

Figure 3.3: RTL schematic of the CPU and L1 cache portions of base architecture.

- *L1-D cache:* The instruction half of the level-one cache, which has a single cycle access time.

These components are interconnected using multiplexers and latches. The base architecture executes instructions in a five-stage pipeline. The pipeline stages are instruction fetch (IF), register file read (R), execute (EX), memory access (MEM) and register file write (W). The parts of the datapath represented by each of these stages is indicated by the labels on the right and left of Figure 3.3.

The CPU executes three classes of instructions: execute, memory and branch. The execution of each class of instruction takes a different route through the datapath. All instructions are fetched from the L1-I cache during the IF stage. Execute and memory instructions can modify the register file and so require all five stages for complete execution. In contrast, branch instructions only require the first three stages for complete execution. Execute instructions compute new data in the execute stage, using either the ALU or the shifter, and do nothing in the MEM-stage. Memory instructions use the EX-stage ALU to perform the address calculation and use the MEM-stage to fetch data from or send data to the L1-D cache. Branch instructions compute the branch condition in the first half of the execute stage using the comparator. The branch condition is then used to control a multiplexer which selects either the branch target address or the next sequential instruction address, which by then, has already been computed in the Next-PC unit.

There are two forwarding paths: EX forward and MEM forward. These provide a way to avoid data hazards between dependent instructions without stopping the flow of instructions through the pipeline (stalling). The EX forward is used to forward data between two instructions that are adjacent in the instruction stream. The MEM forward is used to forward the data between two instructions that are one instruction apart. An instruction that is separated by more than one instruction from an instruction on which it is dependent can read its operands directly from the register file.

Data and control dependencies in the instruction stream create feedback loops in the RTL structure. Each *feedback loop* has a latency which is defined as the

number of CPU cycles it takes to go once around the loop. This latency is equal to the instruction dependency distance. The instruction dependency distance determines the minimum distance between the dependent instructions in the instruction stream that make use of the feedback loop. Dependent instructions that use a particular feedback loop but are closer than the loops' instruction dependency distance must be separated by independent instructions or stall cycles.

Identifying feedback loops in the RTL structure of a microprocessor is important because these loops will be used to interpret the output of in timing analysis algorithms later in this chapter.

In the base architecture there are four types of feedback loop which correspond to four classes of dependencies between instructions. These feedback loops and the instruction dependencies that require them are shown in Figures 3.4 and 3.5. The four types of feedback loop are the EX loop, which has a latency of one cycle; the MEM loop, which has a latency of two cycles; the BR loop which has a latency of two cycles and the RF loop which has a latency of three cycles. The propagation delay of the EX loop, assuming that an ALU instruction is being executed, is the sum of the following components:

- multiplexers MUX_1 or MUX_2
- latches L_6 or L_7
- ALU
- latch L_9,
- multiplexer MUX_4
- interconnect

The propagation delay of the EX loop, assuming that a Shift instruction is being executed, is the sum of delays of the following components:

- multiplexer MUX_3
- latches L_6 or L_7
- Shifter

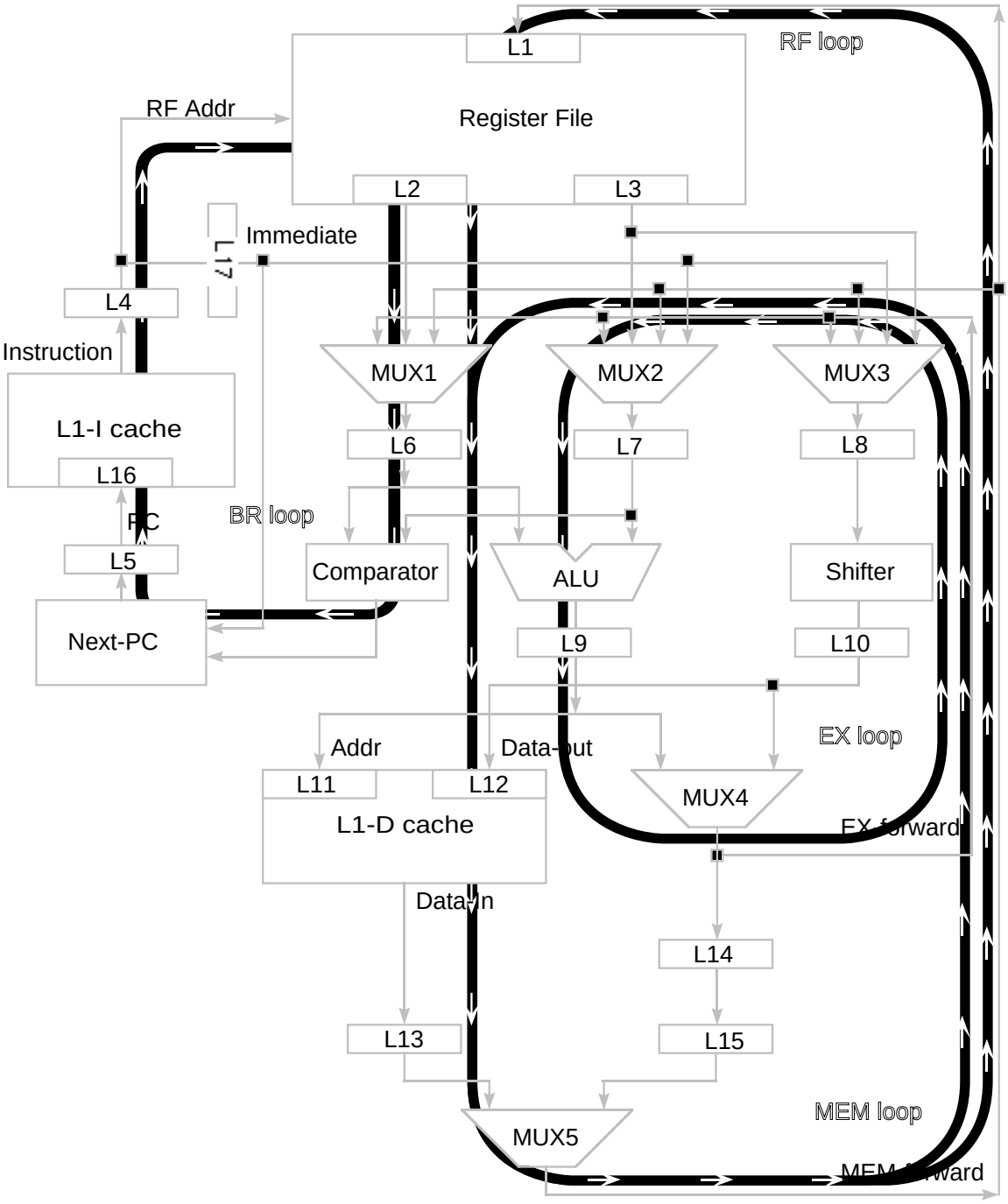

Figure 3.4: The four types of feedback loop.

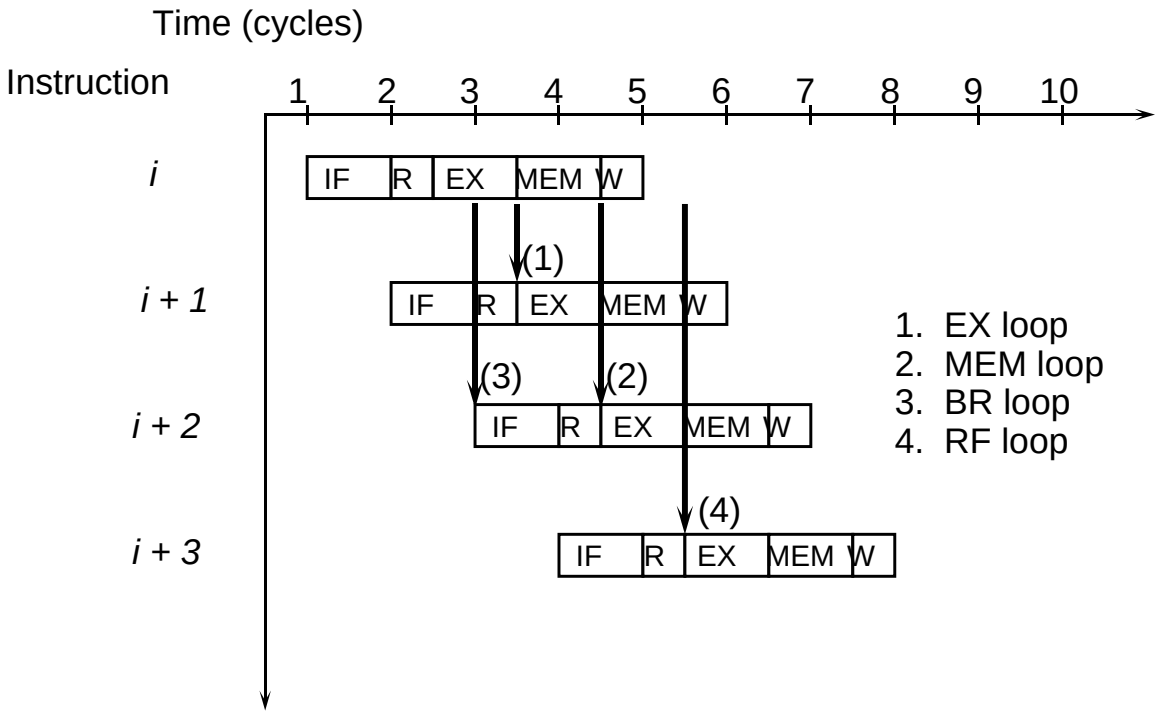

Figure 3.5: Instruction dependencies and their corresponding feedback loop.

- latch L_9
- multiplexer MUX_4
- interconnect

The propagation delay of the MEM loop is the sum of delays of the following components:

- multiplexers MUX_1 or MUX_2
- latches L_6 or L_7
- ALU
- latch L_9
- L1-D cache read
- latch L_{11}
- latch L_{13}
- multiplexer MUX_5

- interconnect

The propagation delay of the BR loop is the sum of delays of the following components:

- next PC logic
- latch L_5
- L1-I cache read
- latch L_4
- RF read
- multiplexers MUX_1 or MUX_2
- latches L_6 or L_7
- comparator
- interconnect

The propagation delay of the RF loop is the sum of delays of the following components:

- RF read
- multiplexers MUX_1 or MUX_2
- latches L_6 or L_7
- ALU
- latch L_9
- L1-D cache read
- latch L_{11}
- multiplexer MUX_5
- RF write
- interconnect

The propagation delay and latency of a feedback loop impose a lower limit on the cycle time of the CPU. Propagation delay and latency also provide the basis of the CPI-t_{CPU} trade-off for CPU pipelines. Increasing the level of pipelining of any of the feedback loops increases the latency and may possibly decrease the CPU cycle time. However, increasing latency of a feedback loop implicitly increases the instruction dependency distance. This, in turn, increases the number of stall cycles, which increase CPI. The exact nature of the trade-off depends on the clocking scheme used in the datapath and the values of propagation delay. Due to the complex nature of the interaction between clocking schemes and propagation delay, a complete analysis of this interaction requires the aid of timing analysis. A timing analysis technique that is capable of analyzing the trade-offs among the latency of the feedback loops, propagation delay and CPU cycle time will be described in the next section.

3.3 Analysis of t_{CPU}

After a review of static timing analysis that points out some of the deficiencies with previous work in this section describes a new timing analysis technique that is ideally suited for use in a multilevel optimization design methodology.

3.3.1 Previouus Timing Analysis Research

In Section 2.3 we saw that rough predictions of the clock frequency can be determined using very simple analytical models. This sort of analysis is useful for comparing different technologies. But we need much more timing accuracy to analyze the CPI-t_{CPU} trade-offs that will be required for high-performance architecture optimization. To provide more detailed timing analysis than a SUSPENS

model researchers have developed timing analyzers. The drawback to using these timing analyzers is that the design must be worked out to a lower level of detail than would be necessary with a SUSPENS model. So they cannot be used in the early stages of computer design. But the accuracy and speed of timing analyzers make them very useful tools in the intermediate and later stages of design. In this

thesis we argue for the use of timing analysis tools for making architectural trade-offs. This is a new idea and we have developed a new type of timing analyzer to make this sort of analysis easier. In the following paragraphs we will briefly review previous timing analysis research to set the stage for the new timing analyzer that is to be described in Chapter 3.

To analyze the timing of the digital circuits used in a modern microprocessor a timing analyzer must be able to analyze circuits that have arbitrary topologies, feedback, level sensitive latches and multiphase clocking schemes. Several "static" timing analysis techniques have been developed to do this. These techniques can be used to accurately predict t_{CPU} given technology parameters and a suitable description of the design, and thus provide an important part of the link between implementation technology and performance. The remainder of this section surveys the work in the area of static timing analysis.

Static timing analysis is a technique for efficiently predicting the timing behavior of digital systems. Static timing analysis achieves its efficiency by ignoring most of the functionality of the circuit; it concerns itself only with circuit propagation delay and synchronization. There are two parts to static timing analysis. The first part is determining the propagation delays of the combinational logic and synchronizing elements (registers, flip-flops, latches) in the circuit. The models used in this delay analysis may be at the register, gate, or transistor levels, or may include a mixture of all three levels. The second part may be to determine the optimal clocking schedule (highest clock frequency) or it may be to validate correct operation with a specific clock schedule. The second part, optimal clocking or validation, is based on the values of propagation delay produced by the delay analysis. Although, these two parts are distinct, many timing analysis tools combine them [Agr82, Dag87, Jou87, Ous85].

Many papers have appeared on static timing analysis of synchronous digital circuits over the last 30 years [Hit82]. However, most of this work has concentrated on circuits with edge-triggered flip-flops as opposed to level-sensitive latches. Circuits with flip-flops are easier to analyze than circuits with level-sensitive latches because flip-flops decouple the input of the latch from the output of the latch. This decoupling

breaks any clocked feedback paths so that, for the purposes of timing analysis, the circuit may be treated as a directed acyclic graph (DAG) and thus analyzed using a breadth-first search algorithm [CLR90]. More recently, researchers have devised timing analysis techniques for circuits with level-triggered latches [Agr82, Che88, DR89, Szy86, Ous85, UT86, WS88].

One of the earliest attempts to analyze the timing of MOS VLSI circuits with level-sensitive latches was made by Agrawal [Agr82]. Agrawal describes a procedure that uses binary-search to find the highest clock frequency that a circuit will operate at without violating latch setup time or hold time. This algorithm requires the user to input the minimum and maximum frequency clock wave forms. The analysis algorithm takes advantage of the fact that level-sensitive latches allow combinational delay elements between them to "share" time between clock phases. This sharing will increase the maximum clock frequency at which the circuit can operate correctly. However, Agrawal only allows sharing among phases that are in the same clock cycle. Jouppi has also used the idea of sharing to analyze MOS VLSI circuits with level sensitive latches that use a two-phase non-overlapping clocking scheme [Jou84]. Jouppi used the term of "borrowing" to describe sharing. Although Jouppi's Timing Verifier (TV) program considers clocking, the main contribution of this work is in the delay analysis of MOS VLSI circuits.

Szymanski developed LEADOUT, which is a static timing analysis tool

for MOS circuits with level-sensitive latches [Szy86]. LEADOUT correctly handles multi-phase clocks. However, the user must specify the clock schedule. A novel feature of LEADOUT is that it compiles the timing constraint equations of the circuit from a causality graph for each clock phase. Once the timing equations have been compiled they can be quickly and repeatedly solved, using simple relaxation, with different clock schedules and delay models. LEADOUT was one of the first timing analysis tool to decouple the delay models from the clocking analysis.

Pearl is a CMOS timing analyzer that, like LEADOUT, begins by constructing a causality graph that represents the dependency of each node in the circuit [Che88]. This causality graph is used to generate a set of linear inequalities that

capture the spacing constraints between all pairs of clock edges used in the circuit. The minimum clock cycle time can be determined by solving the set of inequalities as a linear program. Although it is not necessary to completely specify the duration of the clock phases as in LEADOUT, Pearl requires the ordering of clock edges. Therefore, Perl may not find the minimum clock cycle time if this requires the clock edges to be reordered. Furthermore, in level-sensitive latch circuits that contain feedback, spacing constraints between pairs of clock edges are not sufficient to capture all the timing constraints in the circuit. In such cases, Pearl may produce an invalid cycle time which is too short.

Wallace recognized the distinction between the two parts of timing analysis, and, therefore, the goal of the Abstract Timing Verifier (ATV) program was to develop a timing analysis tool that had no built-in model of delay [WS88]. As such, ATV has been used to analyze transistor level, gate level and register transfer level circuits. ATV uses a dependency graph (causality graph) that supports reference frames as input. These frames of reference are used to analyze circuits in which the clock schedule is unspecified. Using reference frames ATV can generate spacing constraints between pairs of clock edges which could be solved as a linear program as was done in Pearl. Like Pearl, ATV requires the user to specify the ordering of clock edges and thus suffers from the same problems. ATV provides a method for analyzing circuits with level-sensitive latches in which critical paths extend over multiple cycles. The method consists of unrolling the dependency graph a user specified number of clock cycles. However, it is not clear whether this method can be used to analyze circuits with feedback loops that contain level-sensitive latches.

Recently, Dagenais has developed a technique for synthesizing optimal clocking schedules for synchronous circuits [Dag87, DR89]. The technique correctly analyzes circuits with multi-phase clocking and feed-back loops of level-sensitive latches. Dagenais' approach is based on timing intervals. A timing interval is associated with each edge of the clock phases used in the circuit. These intervals are used to generate timing constraints for the circuit. These timing constraints are nonlinear and so cannot be solved using linear programming techniques. An iterative algorithm

is used to find the optimal clocking schedule. The tool that implements this algorithm is called TAMIA. TAMIA analyzes MOS transistor-level circuits. TAMIA is restricted to a single iteration of the algorithm because of the long processing time of the transistor-level delay model. Dagenais does not explore the possibility of compiling the timing constraints so that the compute intensive delay calculation only needs to be performed once.

3.3.2 The $checkT_C$ and $minT_C$ Timing Analysis Tools

$CheckT_C$ and $minT_C$ are timing analysis tools developed by Sakallah, Mudge and Olukotun to analyze and optimize the cycle time of general synchronous circuits [SMO90a, SMO90c, SMO90b]. These tools can be used to answer the following questions about the cycle time of a digital circuit:

- Will the circuit function correctly at a particular cycle time?
- What is the minimum cycle at which the circuit can operate?
- What are the critical circuit paths that limit the cycle time?
- What is the trade-off between cycle time and latency of the critical path?

$MinT_C$ uses a general timing model of synchronous digital circuits to generate timing constraints which simply and precisely capture the conditions for the correct operation of the circuit. These timing constraints can then be used to verify that the circuit functions correctly given a specific clock schedule or they may be used to find the optimal clock schedule (i.e. minimum cycle time).

The $checkT_C$ and $minT_C$ tools have all the benefits and also make up for some of the deficiencies of previous work. These tools properly analyze digital circuits with feedback loops, multi-phase circuits and multi-phase clocking schemes. In addition, there is no built in model of delay so the tools can be used at any design level that has the notion of synchronizer elements (flip-flops or lathes).What distinguishes these tools from previous work is the ability to find the minimum cycle time of these general digital circuits.This ability makes $minT_C$ ideally suited for trade-offs between latency

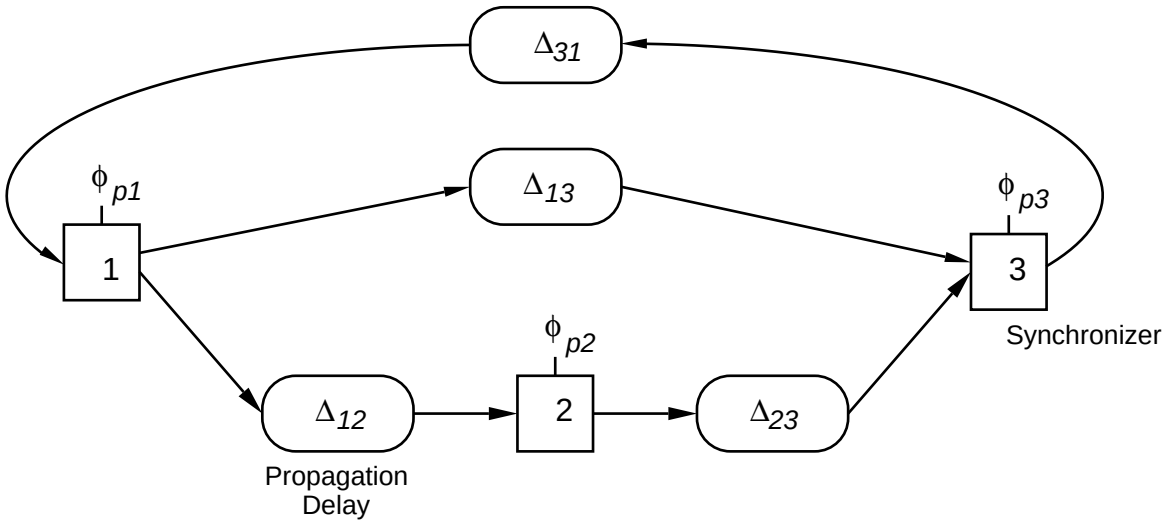

Figure 3.6: The circuit model.

and cycle-time, where the minimum cycle time for each value of latency ensures an accurate analysis of the trade-off. All this is accomplished by the timing tools using a very simple and intuitive timing model.

A Timing Model for Synchronous Circuits

The timing analysis tools are based on a simple circuit model. This circuit model has two components: synchronizer elements and propagation delay elements. Each synchronizer has three ports: a *data-input* port, a *data-output* port and a *clock* port. A circuit is defined by the interconnection of the output ports and input ports of different synchronizers through propagation delay elements and by the assignment of a clock phase signal to each synchronizer clock port. An example circuit is shown in Figure 3.6.

The temporal relationships that are required for the correct operation of a circuit expressed using the circuit model can be divided into three classes:

- Relationships between the different clock phase signals that control the synchronizer clock ports.

- Relationships between the arrival and the departure times of data signals at the input port of a synchronizer and the controlling clock phase signal of the

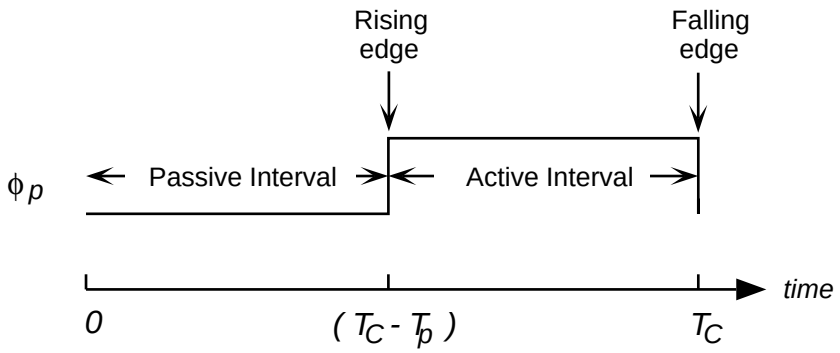

Figure 3.7: The local time frame of phase ϕ_p.

synchronizer.

- Relationships between the propagation delay elements and the arrival and departure times of data signals at the synchronizers.

In the following, these three classes of temporal relationships will be developed into a temporal model for synchronous digital systems. The model is general enough to model both level sensitive and edge-triggered latches. However, in this development we will assume that all synchronizers are level sensitive latches because edge-triggered latches can be treated as special cases of level sensitive latches.

Clock signals are used to control the flow of data signals through latches. Each latch is controlled by a single clock phase of a clock schedule that may contain multiple clock phases. A clock phase ϕ_p, with a phase width of T_p and a cycle time of T_C is shown in Figure 3.7. The local time frame of a clock phase starts at the *falling edge* of the phase and extends from 0 to T_C. This time frame is divided into two time intervals— a *passive* interval followed by an *active interval*. The passive interval starts at the falling edge, ends at the *rising edge*, and has a duration of $T_C - T_p$. The active interval starts at the rising edge and extends to the falling edge and has a duration of T_p. During the active interval the latch is enabled and data signals may propagate from the input port to affect the value of the output port of the latch. At the end of the active interval the signal at the output port is latched and will not change again until the end of the passive interval.

Each clock phase ϕ_p of an arbitrary k-phase $(\phi_1, \phi_2, \ldots, \phi_k)$ clock schedule

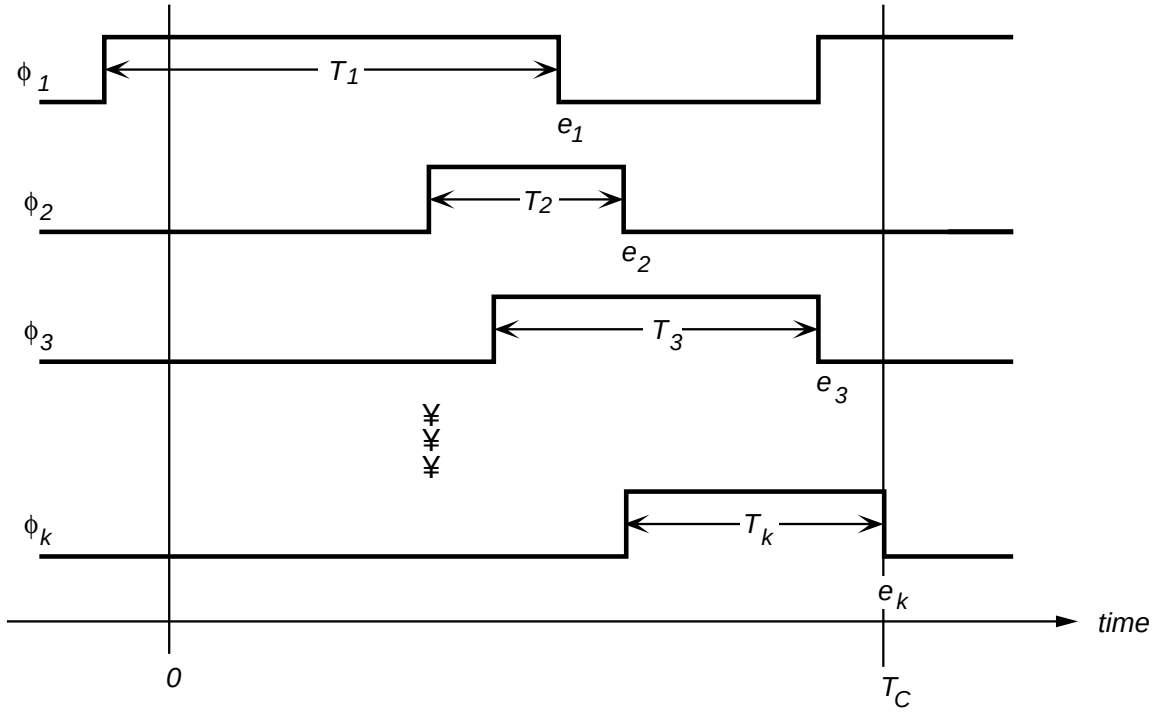

Figure 3.8: A k phase clock. The phase shift E_{12} translates times from ϕ_1 to ϕ_2

.

with a common cycle time T_C may be described by its phase width T_p and its phase ending time e_p. The ending time e_p of a clock phase is the end of its active interval relative to an arbitrarily selected global time phase. Given the description of each clock phase we can define the relationship between the local time frame of any pair of clock phases ϕ_i and ϕ_j. This relationship is defined in terms of a *phase shift operator*:

$$E_{ji} = T_C - [(e_i - e_j) \bmod T_C] \tag{3.1}$$

E_{ij} represents the shift in time necessary to translate times in the local time frame of phase ϕ_i to the local time frame of phase ϕ_j. Figure 3.8 shows an example clock schedule in which ϕ_k is the global clock phase.

A latch i can be characterized by the following five parameters:

- p_i: the clock phase that controls the latch.
- S_i: the setup time before the end of the active interval of p_i during which the signal at the input port must remain stable.

- H_i: the hold time after the end of the active interval during which the signal at the input port must remain stable.
- δ_i: the minimum propagation delay from the input port to the output port.
- Δ_i: the maximum propagation delay from the input port to the output port.

The *arrival* and *departure* times of data at the input port of a latch i are modeled with four variables:

- a_i: the earliest arrival time of a signal transition.
- A_i: the latest arrival time of a signal transition.
- d_i: the earliest departure time of a signal transition.
- D_i: the latest departure time of a signal transition.

These variables are all defined in the local time frame of the clock phase that controls the latch. The synchronization performed by an ideal latch i *i.e.* a latch whose propagation delay is zero may now be specified as a relationship between the arrival times and the departure times of the latch i as follows:

$$d_i = \max(a_i, T_c - T_{p_i}) \tag{3.2}$$

$$D_i = \max(A_i, T_c - T_{p_i}) \tag{3.3}$$

Equation 3.2 specifies that the earliest departure time d_i must occur at the later of the earliest arrival time a_i and the rising edge of ϕ_{p_i}. Equation 3.3 specifies that the latest departure time d_i must occur at the later of the latest arrival time A_i and the rising edge of ϕ_{p_i}.

To ensure that the latch actually latches data properly, there are other constraints on the arrival and departure times of signals at a latch that must be satisfied. To provide enough time to change the state of a latch before the end of the active interval the latest arrival time A_i must occur before the setup time S_i. To ensure that the data in signal does not change before the state of the latch has

stabilized, the earliest arrival time a_i must occur after the latch's hold time H_i. These constraints are modeled by the following inequalities:

$$A_i \leq T_C - S_i \tag{3.4}$$

$$a_i \geq H_i \tag{3.5}$$

At any point in time, the data signal at the input port of a latch can be in one of three different states: *changing*, *stable-new* and *stable-old*. The changing state starts at the earliest arrival time a_i and ends at the latest arrival time A_i and represents the period of time the data signal is unstable. The stable-new state starts at A_i and ends with the falling edge of the clock phase that controls the latch T_{p_i}. The stable-old state starts at the falling edge of the clock and ends at a_i. These states are shown in Figure 3.9 for the three cases of the temporal relationship among T_{p_i}, a_i and A_i.

In the circuit model latches are interconnected by propagation delay elements. Each propagation delay element between a latch j and a latch i is characterized by the following parameters:

- δ_{ji}: the minimum propagation delay of the element.
- Δ_{ji}: the maximum propagation delay of the element.

Given these parameters we can now temporally model the propagation of data between latches j and i by the following equations:

$$a_i = \min_{j \in \mathcal{FI}_i} d_j + \delta_j + \delta_{ji} - E_{p_j p_i} \tag{3.6}$$

$$A_i = \max_{j \in \mathcal{FI}_i} D_j + \Delta_j + \Delta_{ji} - E_{p_j p_i} \tag{3.7}$$

Equation 3.6 expresses the earliest arrival time of a signal at latch a_i as the minimum value for all latches j that fan-in to latch i ($\mathcal{FI}_i$) of the sum of the earliest departure time of latch j and the minimum propagation delay between the latches. Equation 3.7 expresses the latest arrival time of a signal at latch A_i as the maximum value for all latches j that fan-in to latch i ($\mathcal{FI}_i$) of the sum of the latest departure time of latch

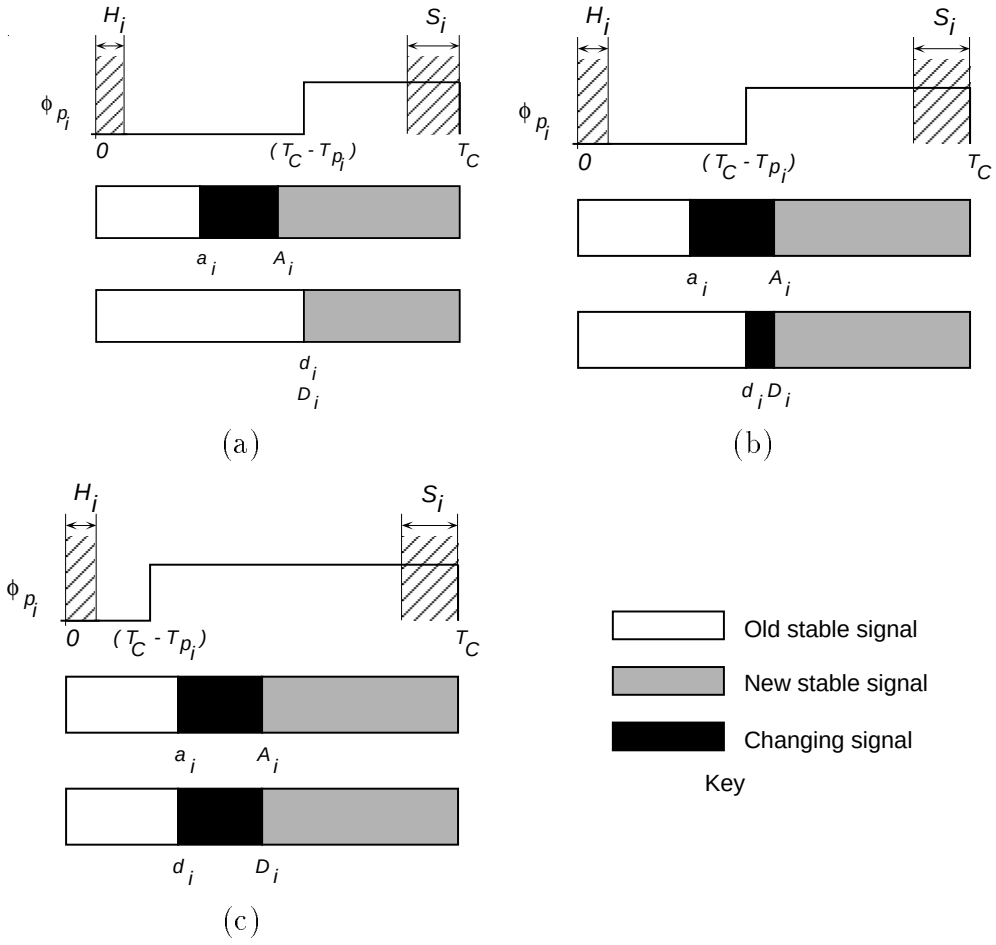

Figure 3.9: The temporal operation of a latch.

j and the maximum propagation delay between the latches. In both equations the phase shift operator $E_{p_j p_i}$ is used to translate the temporal variables of latch j into the time frame of latch i. The delay in (3.6) is referred to as the short path delay while the delay in (3.7) is referred to as the long path delay.

Given a circuit that satisfies the circuit model we can construct a temporal model for this circuit using (3.1)–(3.7). The next section will show how the temporal model of a circuit can be used for to verify that a circuit works correctly given a specific clock schedule (timing verification) or to determine the optimal clock schedule (optimal clocking).

Timing Verification and Optimal Clocking

Timing verification checks that a circuit with a fully specified clock schedule does not violate any setup or hold constraints. The algorithm used to perform this check, $checkT_C$, is described in [SMO90b] and uses a relaxation scheme to find the arrival and departure times of all the latches. The algorithm starts by setting the early departure time d_i of each latch i to its maximum value of $T_C - S_i$ and by setting the late departure time D_i to its minimum value of $T_C - T_{p_i}$. Then the algorithm iteratively applies the synchronization equations (3.2)–(3.3) and the propagation equations (3.6)–(3.7) to all latches until the time values of the latch variables stop changing. The arrival times are then compared with the setup and hold time constraints (3.4)–(3.5) to check for violations. This procedure can also be used to find the *slack* times of (3.4) and (3.5). Slack time is the amount of time that must be added to A_i or subtracted from a_i so that inequalities (3.4) and (3.5) become equations. If the slack time of a latch i is zero, i.e, $a_i = H_i$ or $A_i = T_C - S_i$, then there is a latch j for which $a_i = d_j + \delta_j + \delta_{ji} - E_{p_j p_i}$ or $A_i = D_j + \Delta_j + \Delta_{ji} - E_{p_j p_i}$. The short or long path from latch j to latch i is said to be *critical*. Critical paths may extend through multiple latches and may form feedback loops like the ones in Section 3.2.2. Short critical paths affect the cycle time because they limit the amount of overlap or increase the amount of non-overlap between clock phases through the hold time constraints — the relationship between the falling edge of the phase that

controls the driving latch and the rising edge of the phase that controls the driven latch. The more overlap there is between clock phases the lower the cycle time. Long critical paths affect cycle time by placing a lower bound on time between the rising edge of the phase that controls the driving latch and the falling edge of the phase that controls the driven latch through the setup time constraints.

Given a clock with k phases, the optimal clock schedule for a digital circuit is an assignment of the phases of the clock schedule to the latches and the arrangement of the clock edges of the clock schedule so that the circuit operates at its minimum cycle time. In order for a circuit to operate correctly at its minimum cycle time the clock schedule must ensure that all setup and hold constraints are met, that signals on the critical path are never delayed because they arrive at a latch before the rising edge of the clock, and that the phase widths are as short as possible to limit the effect of the hold time constraints.

Finding the optimal clock schedule of a circuit with feedback loops and level sensitive latches is a difficult problem because the equations are coupled in two ways. Firstly, the arrival and departure times of the latches and thus the critical paths depend on the clock schedule. Secondly, due to the feedback loops it is not clear where to begin analyzing the circuit. Clearly, an iterative optimization method is necessary to find the minimum cycle time that satisfies the set of coupled equations produced by the timing model.

To find the minimum cycle time the variable T_C must be minimized subject to the constraints imposed by (3.1)–(3.7). However, due to the presence of mod, minimum and maximum expressions, these equations are nonlinear. To avoid the computational expense of solving a nonlinear optimization problem, the mod, minimum and maximum expressions can be replaced with linear inequalities. This transforms the nonlinear optimization problem into a linear optimization problem and allows a solution to be found using linear programming [Mur83].

The transformation of the nonlinear timing model into a linear model is as follows. The phase shift operator (3.1) contains a mod expression. To remove this mod expression we must restrict, somewhat, the generality of the model. This is

done by ordering the ending times of the phases relative to the global phase so that $e_1 \leq e2 \leq \ldots \leq e_{k_1} \leq e_k$. With this phase ordering we can transform (3.1) into:

$$E_{ji} = \begin{cases} e_i - e_j & \text{if } e_i > e_j \\ T_C + e_i - e_j & \text{if } e_i \leq e_j \end{cases} \tag{3.8}$$

The implicit phase ordering required by the use of (3.8) may lead the linear programming algorithm to find a cycle time that is not optimal. In such cases a different phase ordering must be considered. The synchronization equations, (3.2)–(3.3), are transformed into the following inequalities:

$$d_i \geq a_i \tag{3.9}$$

$$d_i \geq T_C - T_{p_i} \tag{3.10}$$

$$D_i \geq A_i \tag{3.11}$$

$$D_i \geq d_i \tag{3.12}$$

The propagation equations, (3.6)–(3.7) are transformed into the following inequalities:

$$a_i \leq d_i + \delta_j + \delta_{ji} - E_{p_j p_i}, j \in \mathcal{FI}_i \tag{3.13}$$

$$A_i \geq d_i + \Delta_j + \Delta_{ji} - E_{p_j p_i}, j \in \mathcal{FI}_i \tag{3.14}$$

The transformed timing model can be solved using linear programming [SMO90b]. However, because the minima and maxima have been changed into inequalities, the solution found may violate one or more of the hold constraints. This may happen because the inequalities do not capture an essential characteristic of minimum and maximum functions: that the minimum or maximum of a set must be a member of the set. To see how this can cause a problem, consider a latch i on a critical long path and on a critical short path. The upper bounds of d_i and D_i are both T_C. If D_i is greater than A_i and D_i is greater than $T_C - T_{p_i}$, latch i cannot be on a critical long path, because if it were, we could reduce D_i and thus reduce T_C, therefore D_i must be either equal to A_i or equal to $T_C - T_{p_i}$. In contrast, consider latch j, on the critical short path that fans into latch i. If d_j is greater than a_j and

greater than $T_C - T_{p_j}$, then a_i may satisfy its hold time constraint when in reality it should not. In this case there are no constraints that will force d_i equal to a_i or equal to $T_C - T_{p_i}$. Although this is a problem, it happens rarely in the analysis of real circuits and it is always detectable when it does happen.

The short-path hold violation problem can be completely eliminated by restricting the timing constraints. This is done by assuming the worst case early departure time of $d_i = T_C - T_{p_i}$. This eliminates (3.13) and replaces it with a set of clock constraints like:

$$T_t - T_C + E_{st} \leq \bigwedge_{i,j \in \mathcal{ST}} \delta_j + \delta_{ji} - H_i \tag{3.15}$$

where $\mathcal{ST}$ is the set of all pairs of latches i, j for which clock phase t controls latch j and phase s controls latch i. Using these clock constraints it is possible to prove that solution to the linear timing model will always satisfy all timing constraints [SMO90c]. However, due to the worst case assumption of the early departure time the cycle time may be greater than minimum clock cycle time. Even so, these constraints are not as restrictive as the requirement for non-overlapping clock phases. The two-phase non-overlapping clocking scheme has been used extensively in the design of VLSI digital systems [GD85]. This scheme requires that any two connected latches must be controlled by two non-overlapping clock phases. This requirement makes the short-path propagation equations and the hold time constraints redundant. However, non-overlapping clocks may increase the cycle time.

The timing analysis technique that has been described here has advantages over previous schemes. First and foremost the timing model is simple. The timing model properly analyzes level-sensitive latches as well as edge-triggered flip-flops. It also captures short and long path propagation delays and circuits with feedback. Lastly, there is a separation between timing analysis and the propagation delay analysis on which timing analysis is based. This allows the $minT_C$ timing analysis tool to be used at the transistor, gate or register transfer level. In the next section we apply the timing analysis tool to the RTL model of the GaAs MIPS microprocessor.

3.3.3 Timing Analysis of the Base Architecture

As example of the utility of $minT_C$the latency-t_{CPU} trade-off of the base architecture will be investigated. $MinT_C$ takes as input a textual representation of the circuit model described above and produces the optimal clock schedule given a particular ordering of the clock phases. The result is output produced in both a textual and graphical format. The graphical format shows the clock phases and the state of data at the input port of each latch. Figure 3.10 shows the circuit model that corresponds to the RTL structure in Figure 3.3. The values of the propagation delay elements are calculated using the formulas below. The actual delay values come from circuit simulation using SPICE models of the key datapath elements (register file, ALU, multiplexers) and estimations of on-chip interconnect [Dyk90, HCHU91]. The off-chip MCM delay was calculated using a transmission line model of the MCM interconnect to the 4 KW cache SRAM array [MBB+91].

$$
\begin{aligned}
\Delta_{1\,2}, \Delta_{1\,3}, \Delta_{4\,2}, \Delta_{4\,3} &= \text{register-file read} &&= 1.2\text{ ns}\\
\Delta_{2\,6}, \Delta_{3\,7}, \Delta_{3\,8} &= \text{4-input-mux} &&= 0.4\text{ ns}\\
\Delta_{6\,5}, \Delta_{7\,5} &= \text{comp. + next-PC-mux} &&= 1.5\text{ ns}\\
\Delta_{6\,9}, \Delta_{7\,9} &= \text{ALU-add} &&= 2.0\text{ ns}\\
\Delta_{8\,10} &= \text{shift.} &&= 1.5\text{ ns}\\
\Delta_{9\,11}, \Delta_{11\,13}, &= \text{MCM-interconnect} &&= 0.9\text{ ns}\\
\Delta_{9\,6}, \Delta_{9\,7}, \Delta_{9\,8}, \Delta_{10\,6}, \Delta_{10\,7}, \Delta_{10\,8} &= \text{2-input-mux + 4-input-mux} &&= 0.6\text{ ns}\\
\Delta_{9\,14}, \Delta_{10\,14} &= \text{2-input-mux} &&= 0.2\text{ ns}\\
\Delta_{13\,6}, \Delta_{13\,7}, \Delta_{13\,8}, \Delta_{15\,6}, \Delta_{15\,7}, \Delta_{15\,8}, &= \text{2-input-mux + 4-input-mux} &&= 0.6\text{ ns}\\
\Delta_{14\,15} &= \text{interconnect} &&= 0.1\text{ ns}\\
\Delta_{13\,1}, \Delta_{15\,1} &= \text{2-input-mux} &&= 0.2\text{ ns}\\
\Delta_{13\,6}, \Delta_{13\,7}, \Delta_{13\,8}, \Delta_{15\,6}, \Delta_{15\,7}, \Delta_{15\,8} &= \text{2-input-mux + 4-input-mux} &&= 0.6\text{ ns}\\
\Delta_{5\,16}, \Delta_{16\,4}, &= \text{MCM-interconnect} &&= 0.9\text{ ns}\\
\Delta_{4\,17} &= \text{interconnect} &&= 0.1\text{ ns}
\end{aligned}
$$

It is assumed that the minimum propagation delays are equal to the maximum propagation delays. All latches have an input port to output port propagation delay of 0.2 ns except latches L_{11} and L_{16} which have propagation delay times equal to the

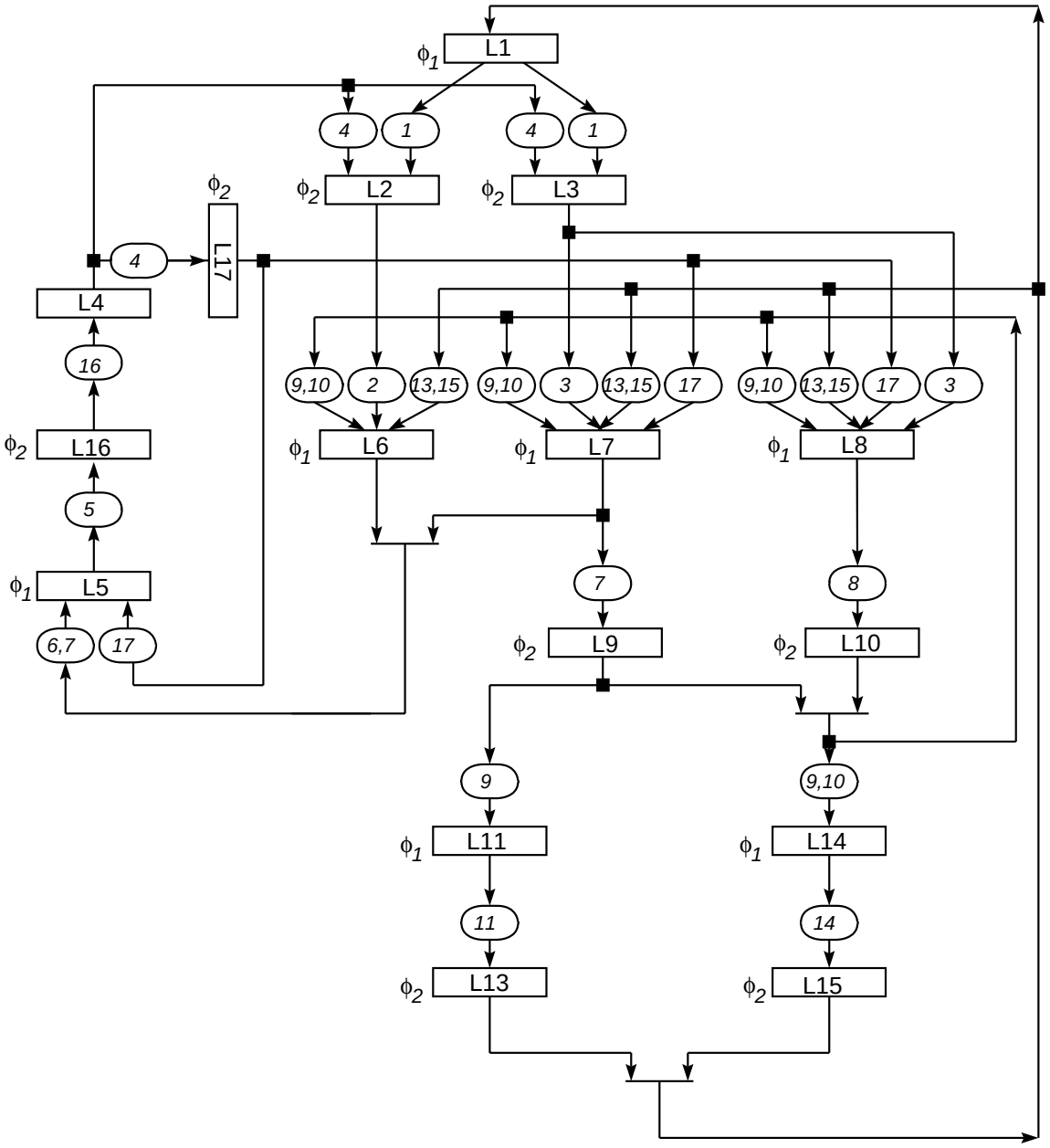

Figure 3.10: The base- architecture circuit model. The numbers in the propagation delay element(s) specify the input latch(s) of the element.

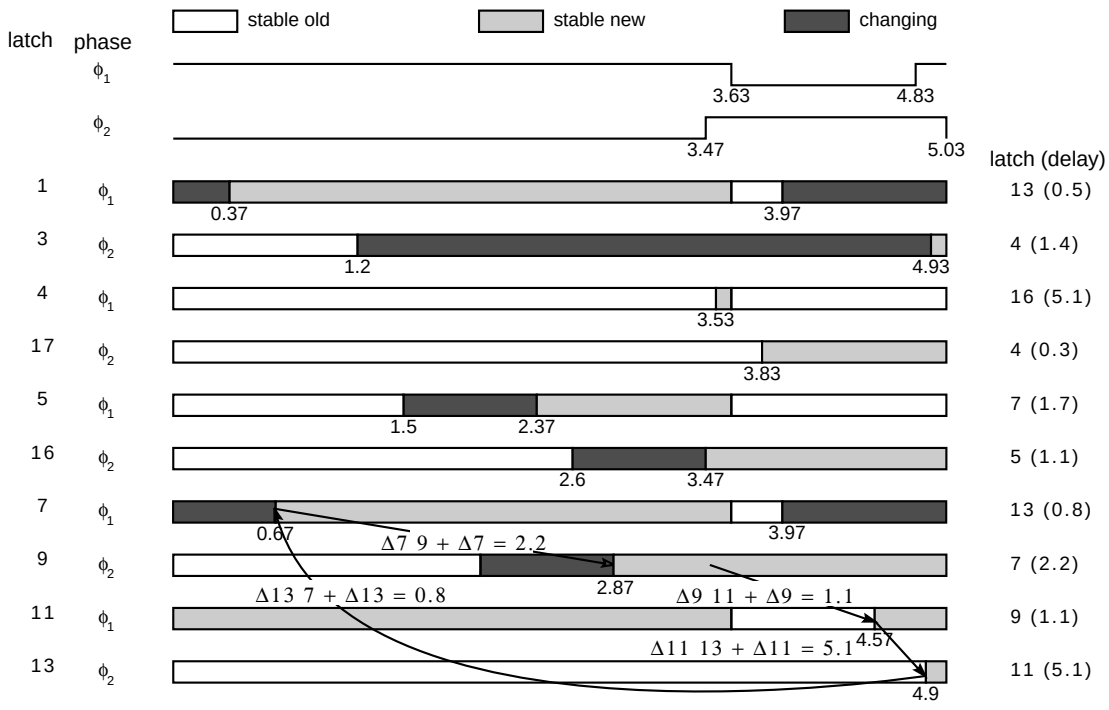

Figure 3.11: Timing of the base architecture datapath. All times are in nanoseconds. The latch names and phase names on the left refer to the signal waveforms. The latch names and delays on the right are the critical long path to the latches labeled on the right.

SRAM access time of 4.2 ns. The setup and hold times of all latches except latch L_1 are 0.1 ns. Latch L_1 has a setup time of 1.0 ns which is the write time of the register file.

The propagation delays shown above and the clock phase assignment of Figure 3.10 (specified in the original design of the datapath) results in the clock schedule and timing waveforms shown in Figure 3.11. This figure does not include some of the latches because either their timing waveforms are identical to one of the latches that is shown or they are not on any critical paths. The cycle time is 5.03 ns. The critical long paths that constrain the minimum clock time are indicated in the figure with arrows between the latest departure time of the source latch and the latest arrival time of the destination latch. These arrows are labeled with the long path delay between these latches. The critical long path corresponds to the MEM loop and includes latches 7, 9, 11, and 13. However, even though latch 9 is on the critical

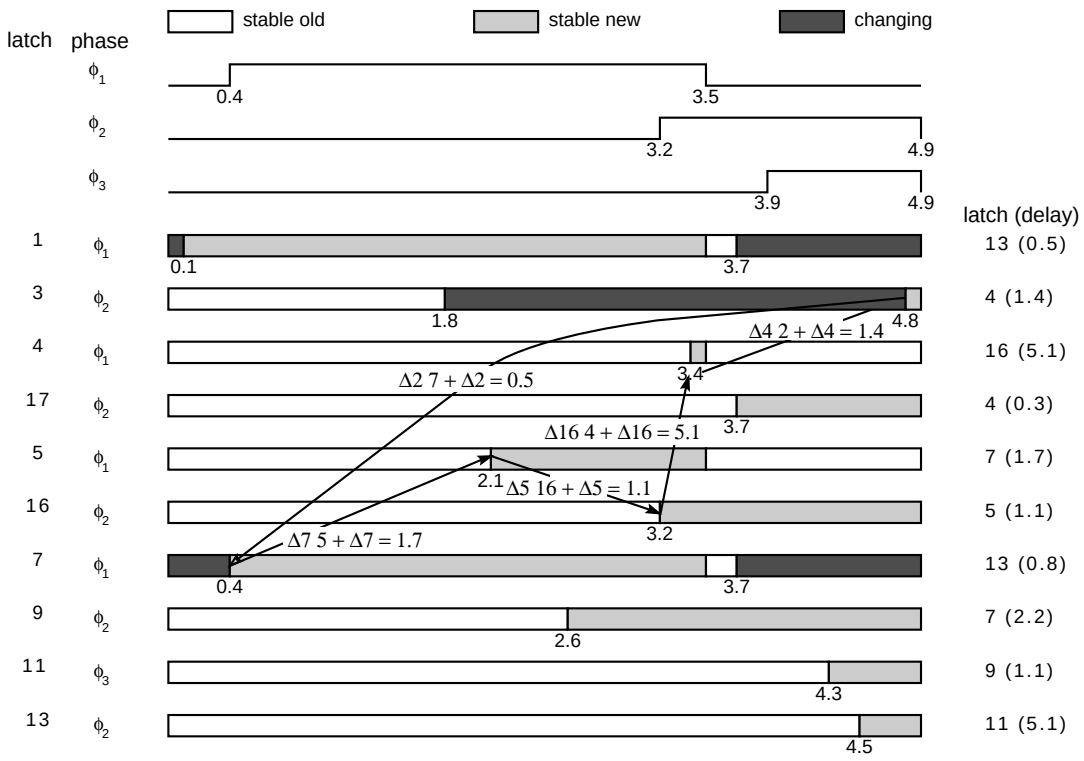

Figure 3.12: Timing after the addition of clock phase for the cache SRAMs.

long path and has a late arrival time of 2.6 ns, its late departure time is delayed until 3.47 ns by the rising edge of ϕ_2 (see Figure 3.11). This delay could be eliminated by extending the rising edge of ϕ_2 to the left by 0.87 ns. However, this cannot be done because it would result in hold time violations.

By introducing another phase to control the L1-D cache address latch (11) into the clock schedule we can reduce the cycle time to 4.9 ns (see Figure 3.12). This reveals that it is, in fact, the branch loop that is the critical long path not the MEM loop as the result of Figure 3.11 indicates. The total delay around this loop is 9.8 ns. This loop has a two cycle latency and so results in a cycle time of $\frac{9.8}{2} = 4.9$ ns.

To further reduce the cycle time the pipeline depth of the MEM loop must be increased. The pipeline-depth of the branch loop must also be increased. This can be done by introducing another latch in the SRAM. The access time of the memory is increased by 0.2 ns to include the extra propagation delay of the latch. The result is a dramatic decrease to a cycle time of 3.33 ns (see Figure 3.14). This cycle time is equal to $\frac{10}{3}$. The cycle can be reduced even further by adding another pipe stage to

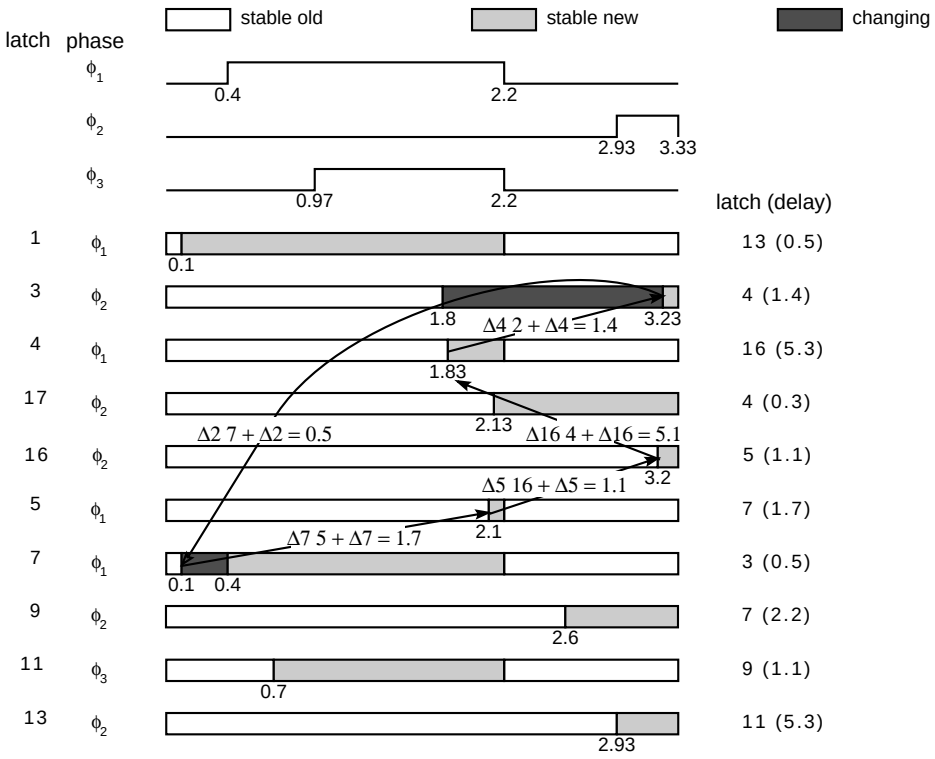

Figure 3.13: Timing with a 2 cycle cache access.

each of the BR and MEM loops. This reduces the cycle time to 3.0 ns and increases the latency of the BR and MEM loops to 4 cycles. The clock schedule that results is shown in Figure 3.14 The cycle time is now limited by the EX loop. Furthermore, the delay in the BR and MEM loops could be increased by 0.45 ns, and 0.6 ns respectively, possibly by increasing the L1 cache size.

Increasing the depth of pipelining in the BR and MEM loops is not done by explicitly changing the circuit model, instead, it is done by modifying the timing model equations and re-solving the linear program (LP). This takes less than a second on a high-performance workstation because the model for the RTL is quite small (72 variables and 120 constraints). However, for circuits at the gate or transistor level the LP may have thousands of variables and constraints and would take a long time to solve. But, it may not be necessary to completely resolve the LP. Using sensitivity or post-optimal analysis the solution to the modified LP can be obtained from a previous optimal solution with far less computation [Mur83].

The clock schedules shown in Figures 3.11–3.14 might be difficult to generate

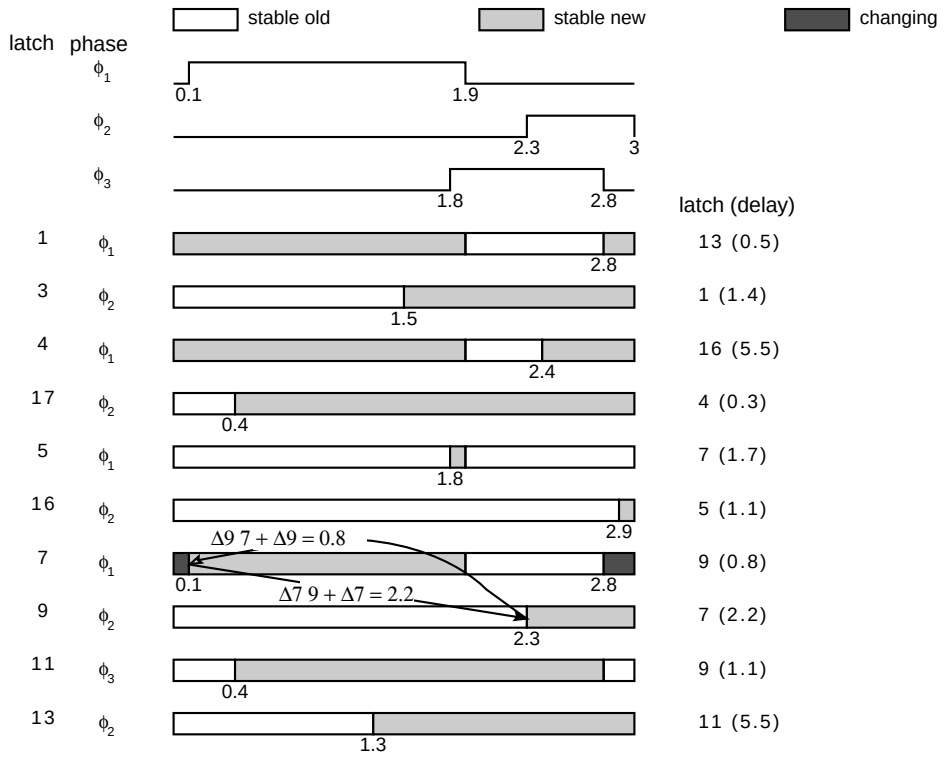

Figure 3.14: Timing with a 3 cycle cache access.

because the clock phases cannot be derived from a single clock phase using delays and inverters. If this is indeed a problem, the timing model may be modified so that the phases width T_{p_i} of each clock phase ϕ_i uses one of two timing variables T_{p_A} or T_{p_B}, where $T_{p_A} + T_{p_B} = T_C$. Adding these constraints may cause the cycle time to be increased. Figure 3.15 shows clock schedule and timing waveforms of the three cycle L1 cache access circuit of Figure 3.14 for which all phases have the same duty cycle or an inverted duty cycle.

Other constraints can be added to the $minT_C$ timing model to model various phenomena. The effect of clock skew can be modeled by adding or subtracting a fixed delay from the propagation delay equations (3.6)–(3.7) depending on whether the clock skew increases or decreases the propagation time between latches. Synchronizing elements that have more complex synchronizing behavior than latches can also be modeled.

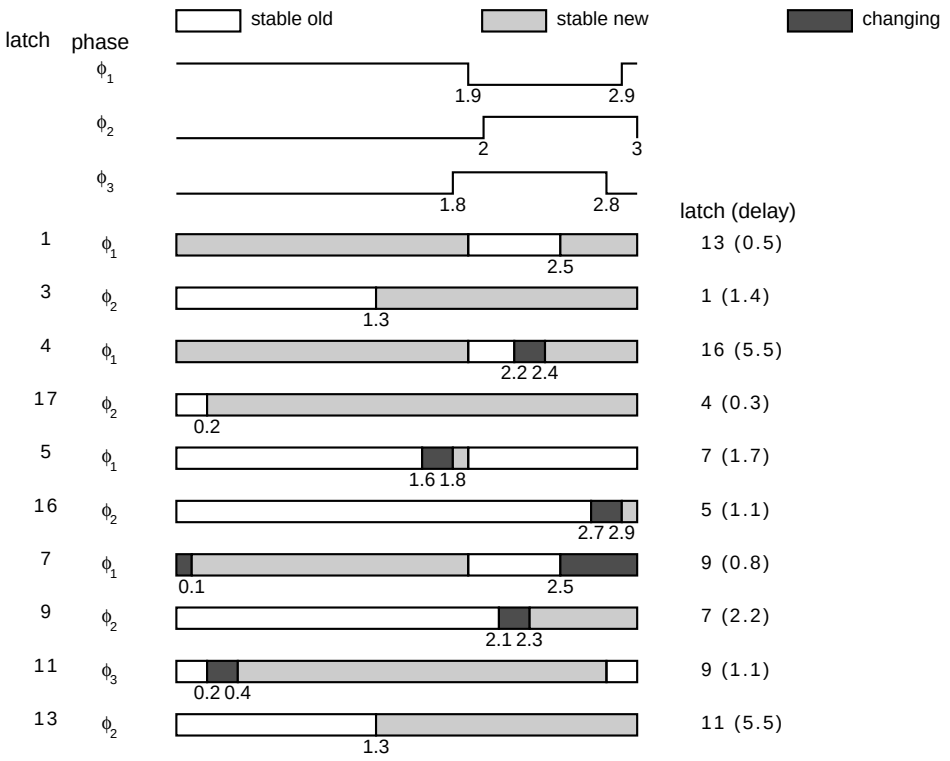

Figure 3.15: Timing when all phases equal to 1 ns or 2 ns.

3.4 Analysis of CPI

For the base architecture and the other architectures that will be investigated in this thesis, CPI can be expressed as:

$$\text{CPI} = 1 + \frac{\text{CPU} - \text{stallcycles} + \text{memory} - \text{stallcycles}}{\text{instructioncount}} \tag{3.16}$$

where CPU-stall cycles are the number of extra cycles spent processing instructions such as loads, branches, and floating-point operations, and memory-stall cycles are the number of extra cycles spent fetching instructions and loading and storing data. To predict the CPI of a particular architecture, a hybrid of trace driven simulation and trace statistics driven performance modeling will be used. The simulators, the benchmark traces, and the CPI of the base architecture will be described in the sections that follow.

3.4.1 Trace Driven Simulation

We have developed a family of trace-driven simulators system to count the number of cycles spent in the CPU and in the memory system. They are all based on the MIPS suite of program performance analysis tools, `pixie` and `pixstats` [MIP88]. The number of cycles that are spent processing instructions in the CPU can be predicted by analyzing each *basic block* [2] using a performance model of individual instruction execution times to predict how many cycles each basic block would take to execute. The number of cycles is multiplied by the number of times the basic block is executed to yield the number of cycles that are spent executing in each basic block. This technique has the advantage that once the basic block execution counts have been collected different architectures can be evaluated quickly by re-analyzing the basic blocks using a different performance model. Furthermore, collecting basic block execution statistics is far less time consuming than full trace-driven simulation.

While this technique works relatively well for components with very small amounts of buffering and concurrency like the MIPS CPU and FPU, it does not work very well for components with large amounts of buffering such as the cache memory system because the effect of such components is spread out over many basic blocks. To predict the performance of cache memory system, full trace-driven simulation must be performed. The number of cycles predicted by the two types of simulator can be added together to provide the total number of cycles spent processing instructions. The drawback in using this technique is that it does not model concurrency between the cache- memory system and CPU. However, a complete trace simulation of the CPU and cache memory system together of the base architecture has shown that for floating-point intensive benchmarks, the benchmarks which are most affected by this inaccuracy, this technique overestimates the number of cycles by at most 10 % [Nag90]. For most integer benchmarks the effect is negligible. Furthermore, most of the inaccuracy is due to an over estimation of CPU cycles. The inaccuracy in cache-memory cycles is less than 1 %.

[2] The instructions between branches.

The cache simulator we have developed, called `cacheUM` is capable of accurately modeling a wide variety of two-level cache memory organizations. The organizational parameters that may be varied include split or unified instruction and data caches, line size, associativity, fetch policy, TLB size, and the virtual-to-physical translation method. The number of cycles it takes to access each level of the memory-hierarchy may also be varied. The simulator also models the effect of a multiprogramming environment. The parameters of the multiprogramming environment that may be varied include the number of processes that may execute concurrently (multiprogramming level) and the time slice (quantum) of CPU time given to each processor.

To prepare for a cache simulation it is necessary to instrument the benchmarks and to create a custom cache simulator. The benchmarks are instrumented to produce address traces by the `pixie` program. `Pixie` takes an object file and augments it with extra instructions at basic block entry points and data reference instructions, so that when the augmented object file is executed, it produces a trace of instruction and data reference addresses. Instrumented object files grow in size by six to seven times. Associated with an instrumented object file is a system call file. System call files enable the simulator to switch among processes when voluntary system call instructions are executed. A system call file that contains the addresses of all system call instructions is generated for each benchmark. A custom cache simulator program is created for each memory organization. This is done by producing a custom C program from a configuration file that describes the parameters of the organization. The resulting cache simulator runs efficiently because it contains only the code necessary to simulate one memory configuration and all the memory configuration parameters are constants. A full multiprogram cache simulation executes at the rate of 240 000 references per second on a MIPS RC3240 (a 15–20 MIPS system), *i.e*, a slow down of a factor of eight.

The parts of cache simulation with `cacheUM` are shown in Figure 3.16, they are: the process controller, the instrumented benchmarks, and the custom cache simulator. The process controller uses UNIX pipes to map the output file descriptor of each benchmark trace to a unique input file descriptor of the cache simulator. Each

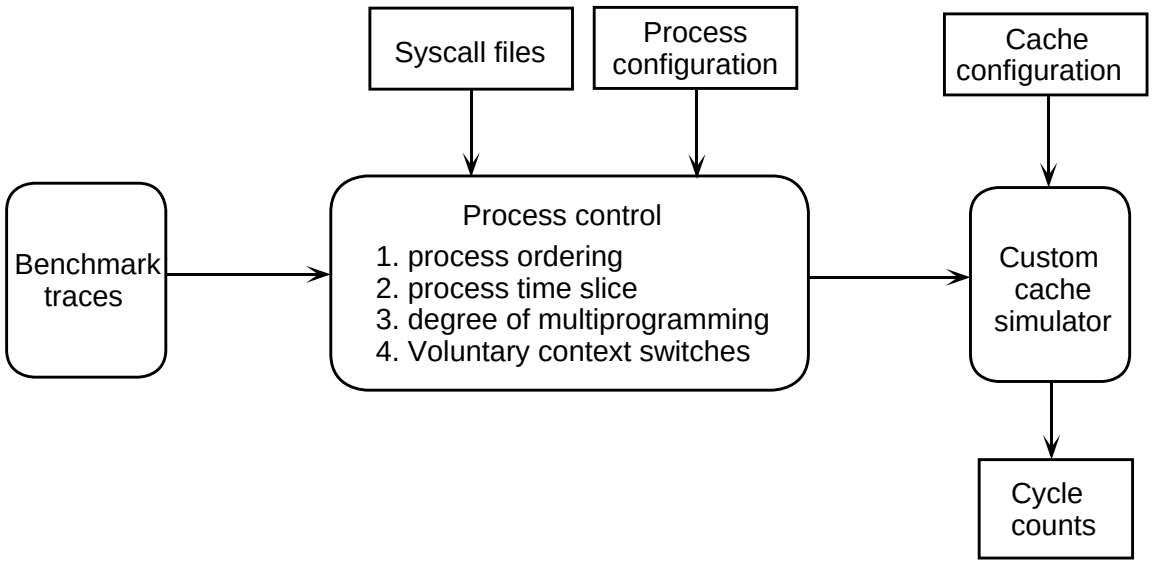

Figure 3.16: Simulation with cacheUM.

benchmark represents a single process. Context switching among benchmark processes is simulated by switching among the input file descriptors from which traces are read. During initialization of a simulation run, the cache simulator reads a process configuration file and the system call files. The process configuration file specifies the multiprogramming level and the order in which the processes will run. During simulation, a context switch is scheduled whenever either the program counter references a system call instruction, or after a time slice has elapsed. The next process that is scheduled for execution is selected using a round-robin schedule. When a benchmark terminates, the next benchmark in the execution order is started. This continues until the execution of all benchmarks is complete.

The trace simulation method described here has three advantages. First, it allows the efficient simulation of very long traces. This technique is able to simulate billions of memory references. Previous techniques have been limited to contiguous traces of under 500 000 reference [AHH89]. Second, this method can realistically simulate a multiprogramming environment. Third, this simulation method allows the same multiprogram simulation trace to be used repeatedly without the need to store the trace. Other methods for generating multiprogramming traces rely on the context switching of a real operating system and therefore are not repeatable [BKW90].

Furthermore, the multiprogramming parameters of this simulation method may be varied without changing the operating system. The disadvantage of this simulation method is that it does not include kernel references between process context switch points.

3.4.2 Benchmark Traces

The importance of a realistic workload to obtaining meaningful predictions of cache performance has been recognized for some time [Smi85a, AHH89]. More recently, the importance of long traces for obtaining accurate performance figures of large caches has also been demonstrated [Sto90, BKW90].

The issue of realistic benchmarks is addressed in this study by using a set of real benchmarks that is representative of the workload of an engineering workstation environment. These benchmarks are listed in Table 3.1. To be even more realistic, an attempt was made to simulate the effect of an interactive user in a graphical windowing environment. This is done by interleaving the X-window application `xlswins` that lists all active windows on a display in between the context switch points of all the other benchmarks. The `xlswins` runs until it executes a system call, at which point the next benchmark is switched in.

A benchmark can be characterized by the average number of instructions between voluntary context switches caused by system calls and by the ratio of working set size to total number of references made by the benchmark. The characteristics of the `xlswins` benchmark are quite different from all the other benchmarks. The average number of instructions between system calls for `xlswins` is 796; the next highest average is for the `tex` benchmark which has an average of 9 236. All other benchmarks have much lower averages. In fact, the `xlswins` accounts for 93 % of all system calls made by the entire workload, yet it creates only 2.2 % of the instruction references. The ratio of working set to trace length is also much higher for `xlswins` than the other benchmarks. The `xlswins` benchmark has a ratio of 0.53 W per instruction reference. The individual benchmark with the next highest ratio is the `gcc` benchmark which has a ratio of 0.0062 W per instruction reference. The ratio for all

Benchmark	Description	Inst. (M)	Loads (% inst.)	Stores (% inst.)	Syscalls (% inst.)	Unique Addr. (KW)
`5diff`	File comparison (I)	218.3	15.3	3.4	305	277.4
`awk`	string matching and processing (I)	209.5	19.0	12.6	101	723.3
`doducd`	Monte Carlo simulation (D)	96.3	31.0	10.0	427	168.1
`espresso`	Logic minimization (I)	238.0	19.9	5.6	17	233.6
`gcc`	C compiler (I)	235.7	23.3	13.8	487	1460.0
`integral`	Numerical Integration (D)	110.5	37.0	10.4	12	423.9
`linpackd`	Linear equation solver (D)	4.0	37.4	19.7	10	33.2
`loops`	First 12 Livermore kernels (D)	275.5	29.3	10.9	3	56.2
`matrix500`	500×500 matrix operations (S)	202.2	24.3	3.5	10	327.6
`nroff`	Text formatting (I)	15.7	22.4	10.8	1701	42.0
`small`	Stanford small benchmarks (I/S)	16.7	19.9	8.8	0	50.4
`spice2g6`	Circuit simulator (S)	297.3	29.8	8.6	395	216.2
`tex`	Typesetting (I)	133.8	30.2	14.2	697	466.0
`wolf33`	Simulated annealing placement (I)	115.4	30.0	7.5	407	200.2
`xlswins`	X-windows application (I)	52.2	22.5	17.7	65294	2672.8
`yacc`	Parser generator (I)	193.9	19.6	2.4	49	197.4
Total		2414.9	24.7	8.7	69915	7548.4

Table 3.1: List of benchmarks that were used to create the multiprogramming traces. Integer benchmarks are denoted by (I), single precision floating point benchmarks by (S), and double precision floating point by (D).

benchmarks excluding `xlswins` is 0.0021 W per instruction reference. The difference in characteristics between `xlswins` and the other benchmarks is analogous to the difference in the behavior of system kernels references to user references [AHH89]. Therefore, the `xlswins` benchmark also serves to approximate the referencing behavior of the missing kernel references. However, because the references of the `xlswins` benchmark and the other benchmarks are uncorrelated, the `xlswins` benchmark is only a crude approximation of system kernel references.

The issue of adequate trace-length actually involves two other issues: ensuring that the cache initialization transient does not distort the miss ratio results and generating enough misses to get an accurate measure of the miss ratio. Researchers have addressed the first issue by defining two types of miss ratio: *cold-start* and *warm-start* [EF78]. The cold-start miss ratio is calculated by counting the number of misses that occur starting from an initially empty cache. The warm-start miss ratio is calculated by first allowing every line in the cache to generate one miss before recording misses for that line. Agarwal has refined the definition of the boundary between the cold-start and warm-start region to include the case where the trace does not contain enough unique addresses to fill up the cache. In this case warm-start is defined to be the point at which the trace has brought its working-set into the cache [AHH89]. A method for detecting when the warm-start region begins is to plot the total number of first time misses to a cache line as a function of trace-length. The knee of this curve indicates the boundary between the cold-start and warm-start region of the trace. Using this method, a cold-start region of a multiprogramming trace with ten processes for a 1 Mb cache is reported to be 600 K references.

In this thesis cold-start miss ratios will be used exclusively. This is done for two reasons. Firstly, the multiprogram traces used in this thesis are three orders of magnitude longer than the cold-start region for large caches reported in [AHH89]. Therefore, the misses of the cold-start region contribute negligibly to the overall miss ratio. Secondly, the multiprogram traces represent the complete execution of the applications that are contained in the trace. In reality, at the beginning of execution an application will have to "warm" the cache by loading its working set into the

cache. Furthermore, there is a possibility the application will be switched out before it leaves the cold-start region. In such cases it may have to reload its working set. Using the cold-start miss ratio with a trace that includes many processes captures all these effects.

The issue of trace length and its effect on the accuracy of miss ratio has been studied by Stone in [Sto90]. Using a Bernoulli process approximation to the cache-miss process, estimates for adequate trace lengths for cache simulation are presented. Stone recommends that 100 misses per cache set are required for reasonably accurate miss ratios. To see how this recommendation affects trace length we will calculate the trace length necessary to accurately predict the miss ratio of the largest cache considered in this thesis. This cache is a 1 MW direct-mapped cache with a line size of 32 W which results in 32 K sets. Assuming a miss ratio of 0.002, to achieve 100 misses per set requires a trace length of 1.6 billion references. The traces used for the experiments in this thesis is 2.5 billion references long.

Another important aspect of realistic trace-driven cache simulation is to ensure that the number of unique addresses in the trace exceeds the size of the caches that are being simulated. Doing this makes it impossible for the cache to hold the working sets of all processes at all times. To satisfy this requirement, in our experiments, the number of unique addresses in the trace used in this thesis is ten times as large as the largest cache size that is simulated.

3.4.3 The Effects of Multiprogramming on Cache Performance

One of the novel features of the trace-driven cache simulator we have developed is its ability to simulate a multiprogramming workload. To see how important this is ant to select the values of multiprogramming level and process switch interval that are appropriate for a system as fast as ours we have used the base architecture and the cache simulator described above to investigate the effect of the multiprogramming environment has on cache performance.

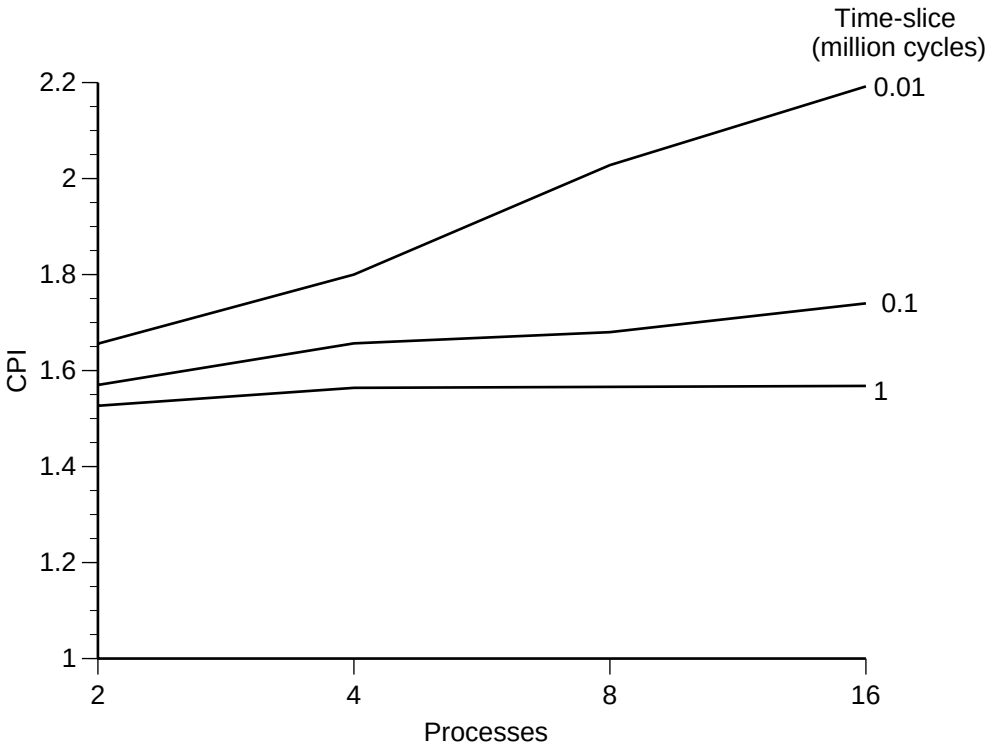

Figure 3.17: The effect of multiprogramming on CPI.

Figure 3.17 shows how CPI varies with the number of processes and the duration of the time slice. Observe that the degradation in CPI due to increasing the number of concurrently executing processes decreases as the duration time-slice increases. The reason for this is that the shorter the time-slice the less time a process has to "warm" the cache with its working set. Therefore, there is a higher probability that a process that is switched out will return to find a portion of its working set in the cache. However, as the number of processes increases, this probability also decreases. This effect diminishes as the time-slice increases because each process brings more of its working set into the cache and more completely displaces the working set of all other processes.

To see more precisely what effect the number of processes has on the cache-hierarchy the miss ratios for the L1 and L2 caches are plotted in Figure 3.18 for a varying number of processes. This figure shows that for a time slice of 0.1 million cycles, performance degrades only slightly as the level of multiprogramming increases. The underlying mechanism for this effect appears to be the following: some of the

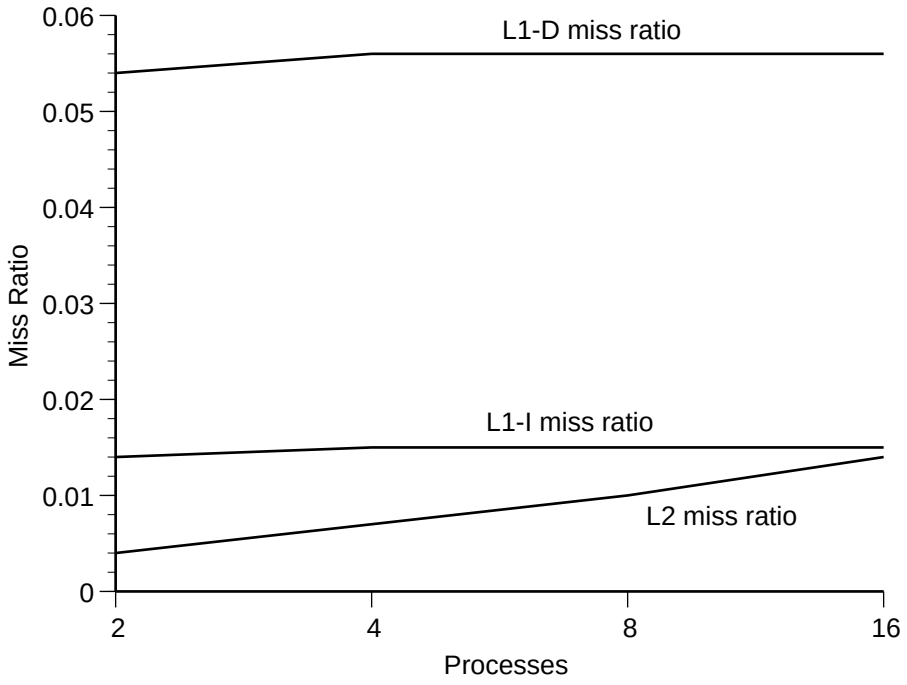

Figure 3.18: The effect of multiprogramming level on cache miss ratio. This data was collected for a time-slice of 0.1 million cycles

lines brought into the cache by a process that is subsequently switched out will be evicted before the process is restarted. The number of lines that are evicted before a process restarts increases with the number of processes between restarts, *i.e.*, as the level of multiprogramming increases. The L1 caches are too small to show this effect in a pronounced way. For instance, the L1-I miss rate changes by only 1 % and the L1-D miss rate changes by only 2 %. However, the L2 cache is big enough to simultaneously contain lines from several processes and so its miss rate changes by 250 %. Fortunately for overall performance, this is 250 % of a very small number ($\approx$ 0.4 %, see Figure 3.18) and so the effect on the overall system CPI is small.

Performance is improved significantly by increasing the time-slice (see Figure 3.19). This is due to the greater opportunity of reusing data that has been brought into the cache. In contrast to multiprogramming level results, the miss ratio of the L1 cache is more adversely affected by the variation in time slice than the L2 cache. The miss ratio of the L1 cache improves by over 50 % as the length of the time slice is varied from 0.01 million cycles while the miss ratio of the L2 cache improves by

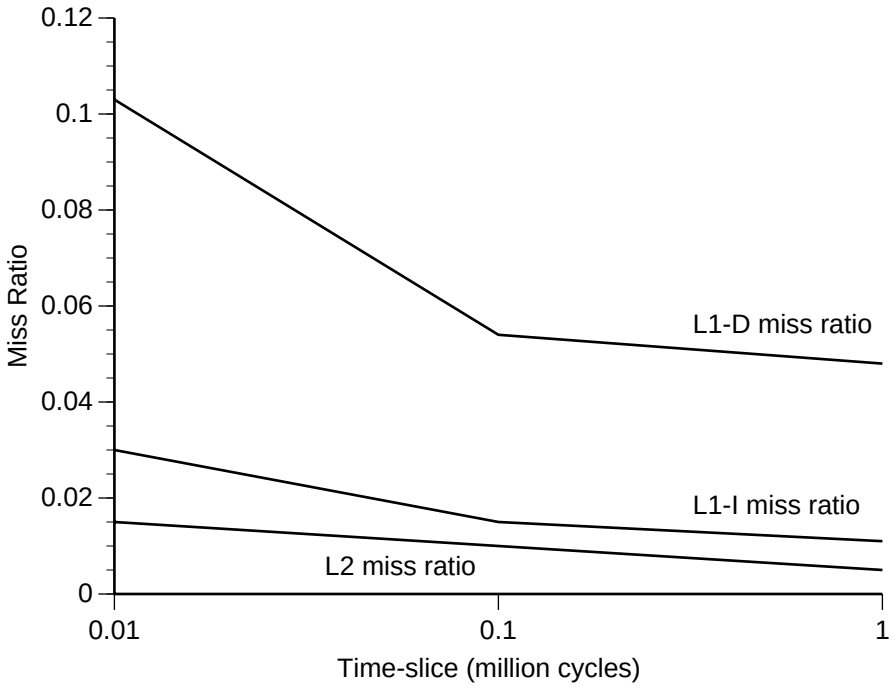

Figure 3.19: The effect of context switch interval level on cache miss ratio. This data was collected with a multiprogramming level of eight processes.

less than 30 % as the time-slice is varied over the same range. The reason for this is that the L2 cache is large enough to capture portions of the working sets of several processes over the period of several context switches. Therefore it is less affected by increases in time slice (time between context switches) than the L1 cache whose contents are completely replaced after each context switch. In other words, if data that are fetched into the L1 cache during a particular time-slice is not reused during that time-slice they will have to be fetched again, but if the same data fetched into the L2 cache they may not have to be fetched again.

It is clear from Figure 3.19 that selecting a time slice that is too short will result in poor cache performance [BKW90]. To guide the selection of a realistic time slice we examined the literature. Clark et al. have investigated the frequency with which context switches and interrupts occur on a VAX 8800 using a hardware monitor [CBK88]. They report an average of 7.7 milliseconds between context switches. This time would translate into 1.9 million cycles for a computer with a 4 nanosecond cycle time. However, this represents a context switch interval that is too long; it ignores

I/0 and timer interrupts which cause operating system kernel code to be executed, thus having a negative effect on cache performance similar to context switching. If we assume that I/0 devices and timer interrupts are unaffected by a shorter CPU cycle time, we could use Clark's figure of 0.9 milliseconds between any interrupt to take these interrupts into account. This time represents 225 000 4 nanosecond cycles. In the rest of the experiments presented in this thesis a time slice of 250 000 cycles for our experiments. This time-slice results in an average of 150 000 cycles between context switches when all system call context switches are also included. This figure excludes the contribution to context switches of the `xlswins` benchmark. It is interesting to note that faster machines may achieve lower cache miss rates because they execute more instructions between context switches. However, the time between context switches will not scale linearly with machine performance. For example a comparison between the VAX 11/780 and VAX 8800 reveals a factor of 3 increase in the time between context switches, but the performance difference between the machines is a factor of five [CBK88]. Furthermore, the trend toward operating systems in which many of the traditional functions of the kernel (virtual memory, file system) are performed by independent user level processes will lead to much shorter context switch intervals [ALBL91].

The architecture supports process identifiers (PIDs) that are included as prefixes to virtual addresses so that each process has a distinct address space. This improves performance by eliminating the need to flush the caches and the translation-lookaside buffer (TLB) after every context switch [Aga88]. The simulator also models this feature. The virtual to physical mapping of addresses is performed using random page assignment.

3.4.4 Base Architecture CPI

The performance of the base architecture is shown in Figure 3.20 for a t_{CPU} of 4 ns. Also shown is the performance loss breakdown from each of the components of the system as different gray levels. The horizontal axis at 1.0 CPI represents the contribution of single cycle instruction execution. The white region labeled by `Proc`.

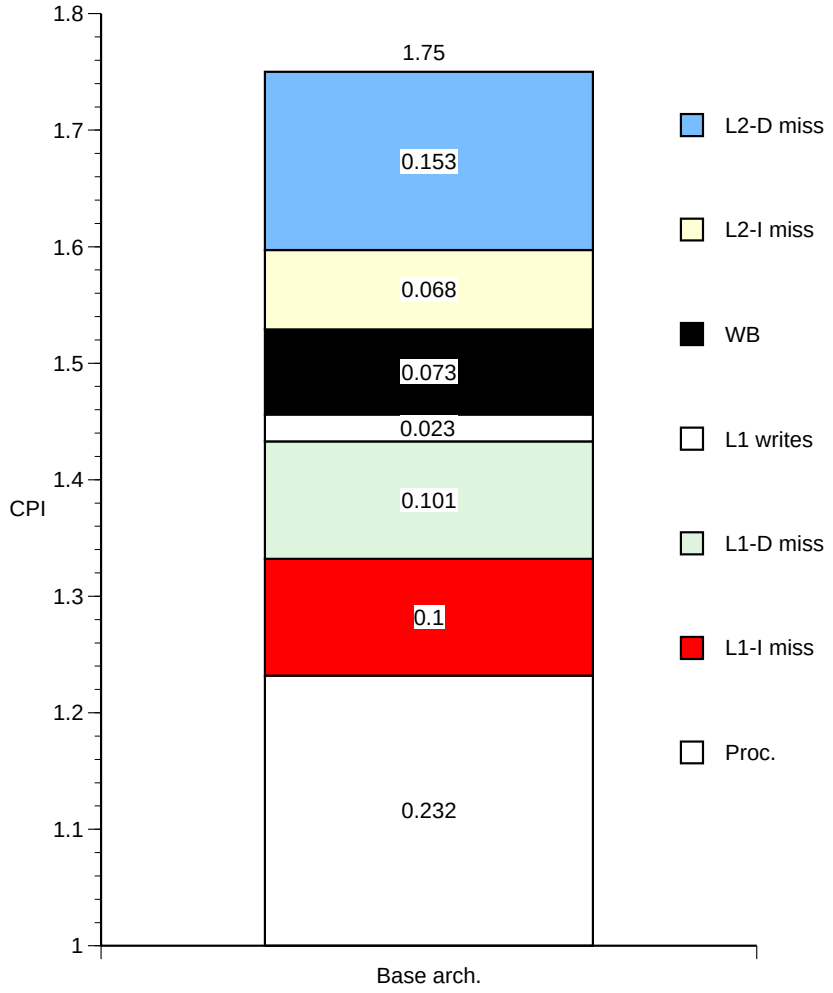

Figure 3.20: Performance losses of the base architecture.

represents the processor stalls resulting from load delays, branch delays and multicycle operations. The portion of the histogram above this region is the contribution to CPI from the cache system which will be the focus of this thesis.

3.5 The Base Architecture Revisited

The CPU in the base architecture is different from the MIPS R3000 in two important respects: 1) it does not have a load-aligner, and 2) it does not have an integer multiplier/divider.

The load aligner was left out of the CPU because analysis of the benchmarks showed that, overall, partial word loads only comprise 2 % of the total number of instructions that are executed. The absence of a load-aligner would require all partial word load instructions to take an extra cycle to align the data using the shifter. This would increase the CPI by the number of partial word loads that are executed. The benchmarks with the most partial word loads are `nroff` with 15 %, `tex` with 6.5 % and `gcc` with 2.6 %. Most other integer benchmarks have far fewer partial word loads and floating point benchmarks have none.

The load aligner has a delay which is roughly 50 % of the ALU add time. This delay is on the MEM loop from the L1-D cache. This extra delay will add from 10-50 % to t_{CPU}. Given the cost in chip area and the increase in cycle time of including a load-aligner against the potential increase in CPI from excluding a load aligner, it was decided to leave it out. An integer multiplier/divider was not included in the CPU because these functions can be executed in fewer cycles using the FPU.

3.6 Conclusions

A multilevel optimization design methodology leads to high performance microprocessor designs by using the characteristics of the implementation technologies to direct decisions at higher levels of design. This design methodology relies on being able to accurately predict the performance of an architecture using TPI. The analysis techniques used to provide this performance prediction are timing analysis

and trace-driven simulation. While neither of these techniques are new, the tools that have been developed for this thesis have new features that improve both their accuracy and versatility compared to previous work. The timing analysis tools use a new timing model. This model makes it possible to find the minimum cycle time of and critical path circuits of a wider class of digital circuits than previously possible. Trace driven simulation has been made more realistic using novel ways to simulate a multiprogramming workload. This simulator was used to explore the effect of multiprogramming parameters on cache performance. The combination of these analysis techniques for architecture optimization are a unique contribution to the field of computer design.

CHAPTER 4

OPTIMIZING THE PERFORMANCE OF L1 CACHES

The organization of the L1 cache has a unique position in the cache hierarchy because it affects both t_{CPU} and CPI directly. This chapter shows the use of the multilevel design optimization methodology to the design of an L1 cache

4.1 The L1 cache optimization problem

The objective of the L1 cache optimization problem is to maximize the performance within the constraints of the implementation technology. To begin the multilevel optimization approach we must specifically characterize how the organization affects TPI.

Restating the equation for TPI as a function of the vector of parameters, P, that characterize the L1 cache organization gives

$$\mathrm{TPI(L1)} = \mathrm{CPI}(P) \times t_{\mathrm{CPU}}(P) \tag{4.1}$$

Elements of P include: L1 cache size, L1 cache associativity, L1 cache access time and L1 cache access latency. Optimization of performance with respect to any of these parameters, say P_i, occurs when

$$\frac{d\mathrm{TPI}}{dP_i} = \frac{d\mathrm{CPI}}{dP_i} t_{\mathrm{CPU}} + \frac{dt_{\mathrm{CPU}}}{dP_i} \mathrm{CPI} = 0$$

Rearranging this equation yields

$$\frac{\mathrm{CPI}'}{\mathrm{CPI}} = -\frac{t'_{\mathrm{CPU}}}{t_{\mathrm{CPU}}}$$

Since TPI and CPI are not analytic functions we make use of the discrete form of the equation instead

$$\frac{\Delta \mathrm{CPI}(P_i)}{\mathrm{CPI}(P_i)} = -\frac{\Delta t_{\mathrm{CPU}}(P_i)}{t_{\mathrm{CPU}}(P_i)} \tag{4.2}$$

This equation states that with respect to L1 cache, performance is maximized when any change in parameter P_i produces an equal relative change in CPI as it does in t_{CPU}.

Given a specific implementation technology, the function $t_{\mathrm{CPU}}(\mathrm{L1})$ can be expressed as

$$t_{\mathrm{CPU}}(P) = f(t_{\mathrm{L1}}, l_{\mathrm{L1}}) \tag{4.3}$$

where t_{L1} is the time it takes to fetch instructions or data from the L1 cache and l_{L1} is the latency of the datapath loop of which the cache is a part. The latency of the datapath loop is determined by the clocking scheme used to control the datapath latches. In general, t_{CPU} is an increasing function of cache access time t_{L1} which can be expressed as

$$t_{\mathrm{L1}} = g(S_{\mathrm{sL1}}, A_{\mathrm{L1}}) \tag{4.4}$$

where S_{L1} is the size of the cache and A_{L1} is the set associativity of the cache. Larger and more set-associative caches have longer access times.

Given a constant latency and transfer time to the next lower level cache (L2) in the memory hierarchy, the function $\mathrm{CPI}(P)$ can be expressed as

$$\mathrm{CPI}(P) = h(m_{\mathrm{L1}}, t_{\mathrm{CPU}}, l_{\mathrm{L1}}) \tag{4.5}$$

where m_{L1} is the miss ratio of the cache. CPI is an increasing function of cache miss ratio. If we hold the latency and transfer times of the L2 cache and the main memory constant, CPI decreases with increasing t_{CPU} because fewer cycles are required to access these levels of memory. The dependence of CPI on l_{L1} is more subtle. In general CPI is an increasing function of l_{L1} because this latency creates instruction execution cycles in which no useful instructions can be executed (delay slots). Branch prediction and reordering of instructions to fill these delay slots are not always successful. Furthermore, aggressive software techniques used to fill branch (BR loop) delay slots

often replicate code, resulting in larger static code size which may lead to a higher L1-I cache miss ratio.

Because of the complex way TPI changes with different organizations of the L1 cache, the functions f, g, and h can only be found by simulation.

4.2 t_{CPU} of the L1 Cache

In this section we will define the function $f(t_{\mathrm{L1}}, l_{\mathrm{L1}})$. The specific timing results of Section 3.3.3 will be generalized so that they can be applied to the class of organizations with the same general RTL structure as the base architecture. Specifically we will estimate the effect of cache size on the cycle time of the base architecture.

In the analysis that follows it is assumed that the EX loop has a latency of one. This assumption is made because the ALU loop has the smallest total delay. All other loops include the cache access time which, for reasonable cache sizes (miss ratio less than 50 %), will always be slower than the ALU in all technologies. Therefore, increasing the latency of the ALU loop would have no effect on the cycle time unless the other loops are heavily pipelined.

To convince the reader that the ALU will normally be faster than memory access, the relative speeds of the ALU add time and memory access time in CMOS technology will be compared. In CMOS, unlike GaAs, it is possible to integrate a reasonable amount of cache on the same chip as the CPU. This eliminates two chip crossing boundaries and greatly reduces the L1 cache access time. To begin the comparison we define the gate delay, D, to be the gate delay measured on a ring-oscillator where each stage has a fanout of three. Given this definition of D, it can be shown that a 64 b group-4 carry lookahead adder has a delay of $12D$ [WF82].

The access time t_{access} of on-chip SRAM can be estimated using the following empirical formula developed by Mark Johnson

$$t_{\mathrm{access}} = D(1 + 1.5 \log_2 N) \tag{4.6}$$

where N is the number of bits in the SRAM [Joh91a]. The rationale for this equation is as follows. The optimum aspect ratio for an SRAM is one [Wan89]. Doubling

both sides of a square array quadruples the number of bits in the SRAM and adds approximately $3D$ to the access time due to longer wires, extra buffering and extra levels of decoding logic. Therefore, doubling the size of the cache increases the access time by $\frac{3D}{2} = 1.5D$. The constant $1D$ term accounts for the minimum delay of the sense amplifier and wordline driver circuits.

We can use (4.6) to define t_{L1} for on-chip direct-mapped caches. The SRAM size with an access time of $12D$, equal to the delay of the adder in an ALU, is between 128 and 256 b. A cache of this size is far too small, especially considering that a 32×32 register file has 1024 b. The smallest cache of reasonable size is 16 Kb (2 KB) and has an access time of $22D$. Thus, even in the case of CMOS where the cache can be placed on the same chip as the ALU, the EX loop is still the fastest path.

Chapter 2 showed that using GaAs technology it is not possible to place the L1 cache on the same chip as the CPU, because a chip of this size has a power consumption that is too high and a yield that is too low to be practical. Therefore the cache must be constructed of individual cache SRAMs that are placed on an MCM. In this arrangement the chip-to-chip delay of the MCM from the CPU to the cache SRAMs, which is denoted by t_{MCM}, is an appreciable component of t_{L1}. The equation for t_{L1} that includes the round trip delay t_{L1} from the CPU to the cache and back is

$$t_{\mathrm{L1}} = t_{\mathrm{access}} + 2t_{\mathrm{MCM}} \tag{4.7}$$

where t_{access} is defined by (4.6).

In general t_{MCM} is dependent upon the electrical characteristics of the MCM interconnect (R, L and C) and the longest distance from the CPU to any cache SRAM. Given n, the number of SRAM chips in the L1-I or L1-D cache, t_{MCM} can be approximated by the following linear equation

$$t_{\mathrm{MCM}} = k_0 + k_1 n \tag{4.8}$$

where k_0 is a constant term associated with the delay of the off-chip drivers and receivers and k_1 is a linear coefficient that represents the additional delay per chip. If n is the number of chips in the L1-I or L1-D cache, then to minimize interconnection

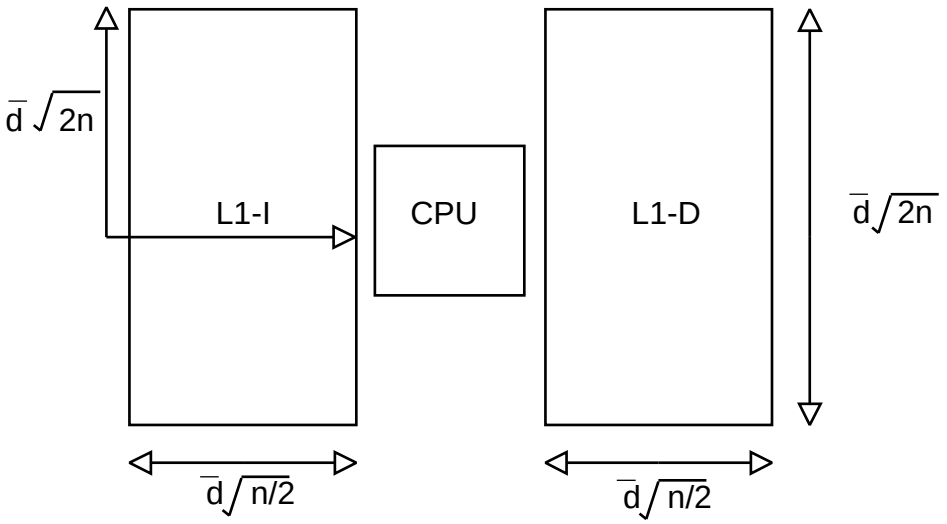

Figure 4.1: The minimum delay arrangement of $2n$ L1 cache SRAM chips.

delay these chips should be arranged as closely as possible to a $\sqrt{n/2} \times \sqrt{2n}$ rectangle as shown in Figure 4.1. If the CPU is placed in the middle of the long side of this rectangle, the maximum length l of a wire from the CPU to any chip is $\overline{d}\sqrt{2n}$ where $\overline{d}$ is the average chip pitch. More specifically, the term $\overline{d}$ is defined as the average of the horizontal h and vertical v pitches of the chip, where h and v include the width of an adjacent wiring channel and any extra space necessary to connect the chip to the MCM.

The value of the linear coefficient can be expressed as

$$k_1 = Z_0 C_{bond} + 2\overline{d}^2 R_{\text{MCM}} C_{\text{MCM}} \tag{4.9}$$

where Z_0 is the characteristic impedance of the MCM interconnect and R_{MCM} and C_{MCM} are the resistance and capacitance per unit length of interconnect. This equation is a modified form of an equation for the packaging delay of interconnect presented in [Bak90]. The first term of (4.9) is the delay due to parasitic capacitance of the bonding method and the pad that connects the chip to the MCM. The second term is the distributed RC delay of the MCM interconnection lines and is proportional to the square of the length of the MCM interconnect that is being driven. However, this length is proportional to the square root of the number of chips in the cache n, making the second term linear in n. Equation (4.9) assumes the interconnect is quite

Parameter	Value
R_{MCM}	1.07 Ω/mm
L_{MCM}	0.272 nH/mm
C_{MCM}	0.108 pF/mm
$R_{\text{MCM}}C_{\text{MCM}}$	0.116 ps/mm^2
Z_0	50 Ω
C_{bond}	0.90 pF

Table 4.1: MCM parameters.

lossy and so neglects any delay from transmission line behavior.

To validate the expression for k_1 and to get an accurate measure of the interconnect delay of the MCM, layouts for cache with 4, 6, 11, and 21 chips were simulated with HSPICE [Met90]. The details of these simulations are presented in [KSH+91]. These simulations assumed an SRAM chip with dimensions of 3.8 mm × 6.8 mm and an MCM with a wire pitch of 40 μm. A wire pitch of 40 μm results in a 32 b wiring channel that is 1.3 mm wide. This in turn results in a $\overline{d}$ of 6.6 mm. The other parameters that are necessary to calculate k_1 are listed in Table 4.1. Combining them yields

$$k_1 = (50\Omega)(0.9\,\text{pF}) + 2(6.6\,\text{mm})^2(0.116\,\text{ps/mm}^2) \quad (4.10)$$

$$= (45\,\text{ps}) + (10.1\,\text{ps}) \quad (4.11)$$

$$= 55.1\,\text{ps} \quad (4.12)$$

From (4.11) it is clear that most of the additional delay that comes from adding extra cache chips is due to the parasitic capacitance of C_{bond}. Therefore, reducing C_{bond} would provide the most significant reduction in t_{MCM}. Unfortunately, the values of C_{bond} assume flip-chip bonding which is the highest performance bonding method that presently exists [Bak90], and there is little that can be done to reduce C_{bond} below the value of 0.90 pF. Given this situation, the parameters of the MCM interconnect R_{MCM} and C_{MCM} must be reduced in order to reduce the MCM delay.

Figure 4.2 plots the results of the simulations which correspond to values of n of 4, 6, 11 and 21. A least- squares linear curve fit to these points has parameters

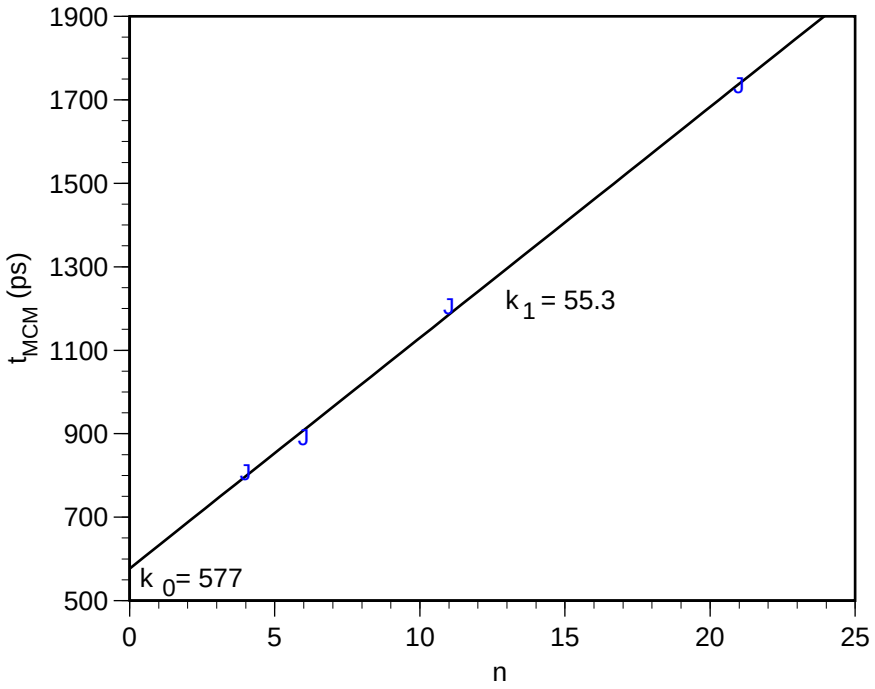

Figure 4.2: t_{MCM} versus the number of L1 cache chips (n).

$k_0 = 577\,\mathrm{ps}$ and $k_1 = 55.3\,\mathrm{ps}$. The value of k_1 is very close to the value of k_1 calculated in (4.12), validating expression (4.9) for k_1.

Equation (4.7) can now be rewritten as

$$t_{\mathrm{L1}} = (1 + 1.5B)D + \frac{4S_{\mathrm{L1}}}{B}\left[Z_0 C_{\mathrm{bond}} + 2\left(\overline{d}_{15}\sqrt{2}^{(B-15)}\right)^2 R_{\mathrm{MCM}} C_{\mathrm{MCM}}\right] + 2k_0 \quad (4.13)$$

where $B = log_2 N$ and $\overline{d}_{15}$ is the chip pitch for a $2^{15} = 16\,\mathrm{Kb}$ chip. The term $\left(\overline{d}_{15}\sqrt{2}^{(B-15)}\right)$ assumes that the chip pitch grows or shrinks by a factor of $\sqrt{2}$ as the number of bits in the chip grows or shrinks by a factor of 2. We could take the derivative of (4.13) with respect to B and set the result equal to zero in order to obtain the chip size that minimizes t_{L1} for a given cache size S_{L1}, however, it is more interesting to plot the equation in order to see the trend that develops as the chip size is varied.

Figure 4.3 is a plot of (4.13) for a cache size 4 KW (16 KB), which is the size used in the base architecture. The plot shows that t_{access} increases linearly and t_{MCM} decreases exponentially with each doubling of SRAM chip size. The linear increase in t_{access} is a direct consequence of (4.6). The explanation of the exponential decrease

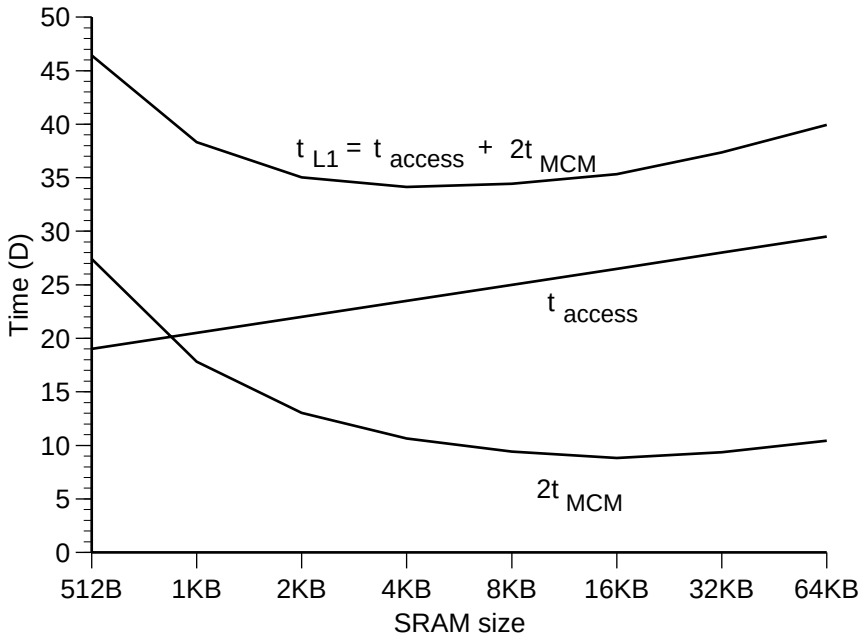

Figure 4.3: A plot of t_{L1} versus SRAM chip size of $S_{\mathrm{L1}} = 4\,\mathrm{KW}$.

in t_{MCM} is more involved. The main reason for the decrease in t_{MCM} is that fewer chips are required to implement the cache as the SRAM chip size increases. Although larger SRAM chips have a longer chip pitch $\overline{d}$, which tends to increase k_1, the largest component of k_1 is $Z_0 C_{\mathrm{bond}}$ which is not dependent on $\overline{d}$. The net effect of these factors is a decrease in t_{MCM} as SRAM chip size increases.

The L1 access time reaches a minimum between SRAM chip sizes of 4 KB and 8 KB. From this, it is clear that larger SRAM chips do not necessarily offer better overall access time. On the contrary, smaller SRAM chips coupled with MCM packaging can achieve lower overall access times. Furthermore, we shall see that small SRAM chips with fast access times have other advantages.

To investigate how the t_{L1} versus SRAM chip size curve changes with the larger sizes of the L1 cache (S_{L1}), Figure 4.4 was plotted. Values of S_{L1} from 4 KW to 32 KW are shown. As the cache size grows the optimum cache access time moves to larger SRAM sizes. However, the optimum point is not pronounced and an SRAM size of 4 KB represents a good choice for all cache sizes less than 32KW.

Equation (4.13) defines function g (4.4) for an MCM based direct-mapped L1 cache. This equation can be combined with the latency of the BR loop for L1-I

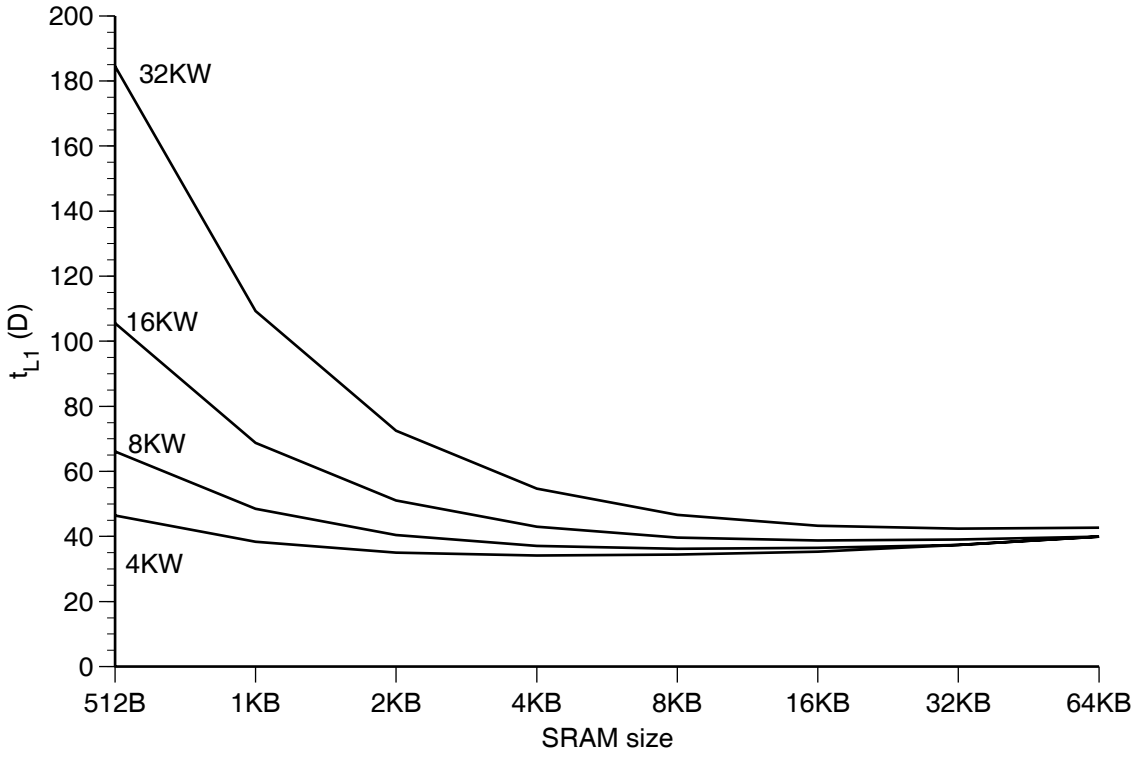

Figure 4.4: A plot of t_{L1} versus SRAM chip size.

and the latency of the MEM loop for L1-D to determine the function f in (4.3) for both L1-I and L1-D. We will consider latency values for these loops of two, three, and four cycles. These latency values correspond to L1 cache delays of one, two, and three cycles. The structure of the L1 cache datapath is shown in Figure 4.5. All memory configurations, regardless of latency, will use this structure. In the MCM L1 cache analysis the latency of the BR loop and MEM loop will be varied by changing the clock schedule. The way in which this is accomplished is shown in Figure 4.6

When the CPU datapath is analyzed with $minT_C$, using the L1 cache access times from Figure 4.3 and the GaAs CPU datapath parameters listed in Tables 4.2 and 4.3, the plot shown in Figure 4.7 results. The value of D used to generate these figures is 150 ps.

The BR loop is the critical path for latency values of two and three because it has a larger amount of delay than the MEM loop. When the latency of the BR loop is two, the minimum cycle time occurs with an SRAM size of 4 KB. This is the same as the size with the lowest access time in Figure 4.3. However, when the latency

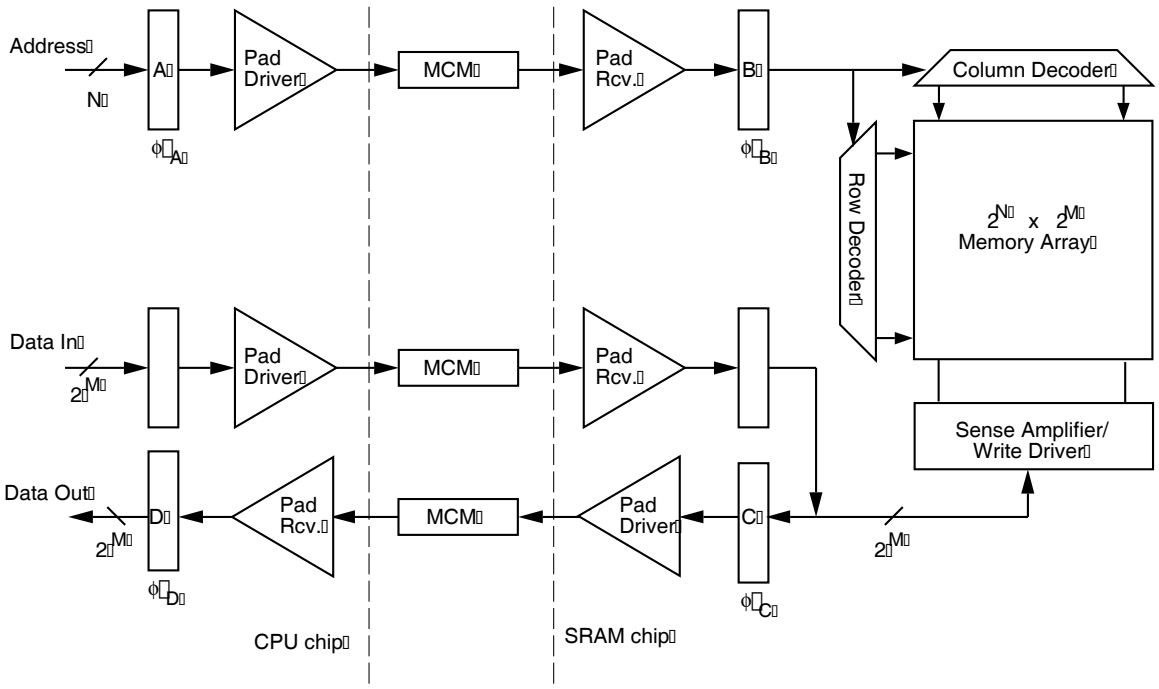

Figure 4.5: The L1 cache datapath. The latches used in an SRAM read cycle are labeled by the letters A, B, C and D. A is the CPU address latch, B is the SRAM address latch, C is the SRAM data output latch and D is the CPU data input latch.

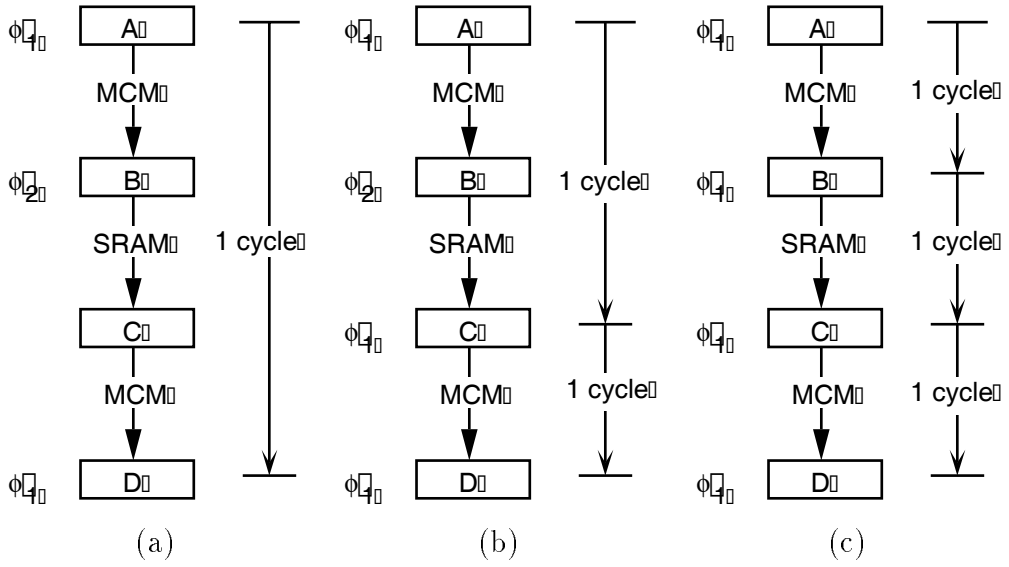

Figure 4.6: Three clock schedules for L1 cache access. (a) one cycle access, (b) two cycle access and (c) three cycle access. The clock phase assignments are examples. Analysis of a specific circuit with $minT_C$ will provide the phase assignment with the optimum cycle time. As shown in Figure 3.12, a special memory clock phase may be necessary to reach the optimal cycle time.

Component	Delay
ALU	$14D$
Register-read	$9D$
Register-write	$9D$
Multiplexor	$(1 + \lceil log_2 \text{inputs} \rceil)\, D$
Comparator	$9D$
Latch	$2D$

Table 4.2: Delay of components of the CPU datapath measured in gate delay D.

Loop	Delay
EX loop	$23D$
BR loop	$(34 + t_{\text{L1-I}})D$
MEM loop	$(29 + t_{\text{L1-D}})D$

Table 4.3: The delay of each loop measured in gate delay D.

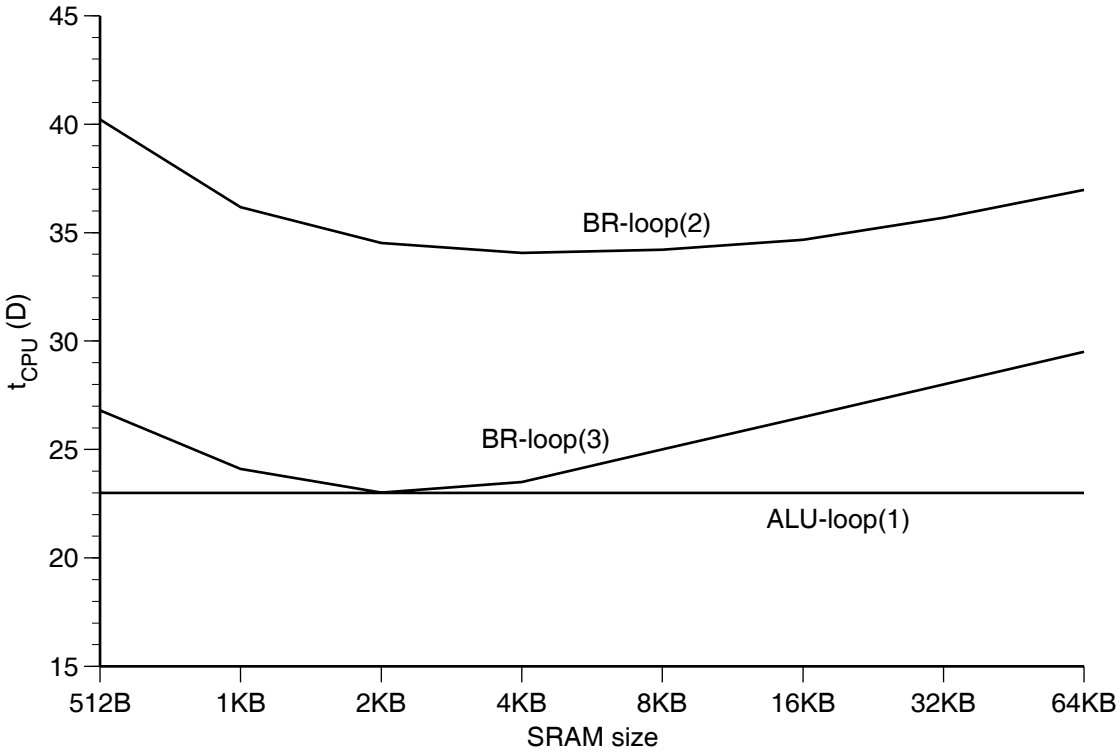

Figure 4.7: A plot of t_{CPU} in units of gate delay (D) versus SRAM chip size for $S_{\text{L1}} = 4\,\text{KW}$. The number in parentheses is the latency of the loop.

is increased to three the minimum cycle time changes to 2 KB. The reason for this is that the longest propagation delay in the circuit places a lower bound on the cycle time. This will always be the case unless the circuit is wave-pipelined [WMF89].

The lower bound on t_{CPU} imposed by the SRAM access time sets the t_{CPU} for SRAM sizes larger than 2 KW for a BR loop latency of three. Therefore, increasing the latency of the BR loop will only reduce the CPU cycle time for SRAM sizes of 2 KW and smaller. For these sizes the increase in t_{MCM} and the decrease in t_{access} equalizes the delay between the latches and makes it possible for deeper pipelining to reduce the cycle time. However, it is not necessary to increase the latency of the BR loop before the benefits of a short SRAM access time can be observed. Even for a latency of three, a 4 KW cache implemented from 2 KB SRAMs achieves a lower cycle time than one implemented from 8 KB SRAMs, despite the fact that the total access time of the cache made from the 2 KB SRAMs is greater than the total access time of the cache made from the 8 KB SRAMs.

To reduce the lower bound on t_{CPU} of t_{access}, it is possible to move the latch labeled A in Figure 4.5 to a position after the row and column decoders. Doing this will increase the number of bits that must be latched in the SRAM and results in a non-standard SRAM design. However, if a custom SRAM is being designed for an on- chip cache, the reduction in t_{CPU} could make this SRAM design worthwhile.

In the remainder of the analysis of L1 caches the cache will be implemented from 4 KB SRAMs because 4 KB represents a good compromise between the shorter access times offered by smaller SRAMs and the short overall L1 cache access times offered by larger SRAMs. Shorter L1 cache access time becomes important when the cache is increased beyond 4 KW. For cache sizes greater than 16 KW Figure 4.4 shows that SRAMs larger than 4 KB provide lower t_{L1} times. However, larger SRAMs in GaAs are difficult to fabricate, have poor yield, and consume relatively large amounts of power. Thus 4 KB represents a good compromise. In CMOS the choice of SRAM size might be different, but the same sort of analysis to quantify the relationship between speed and size is applicable.

Given an SRAM size of 4 KB, Figure 4.8 plots the minimum cycle time for L1

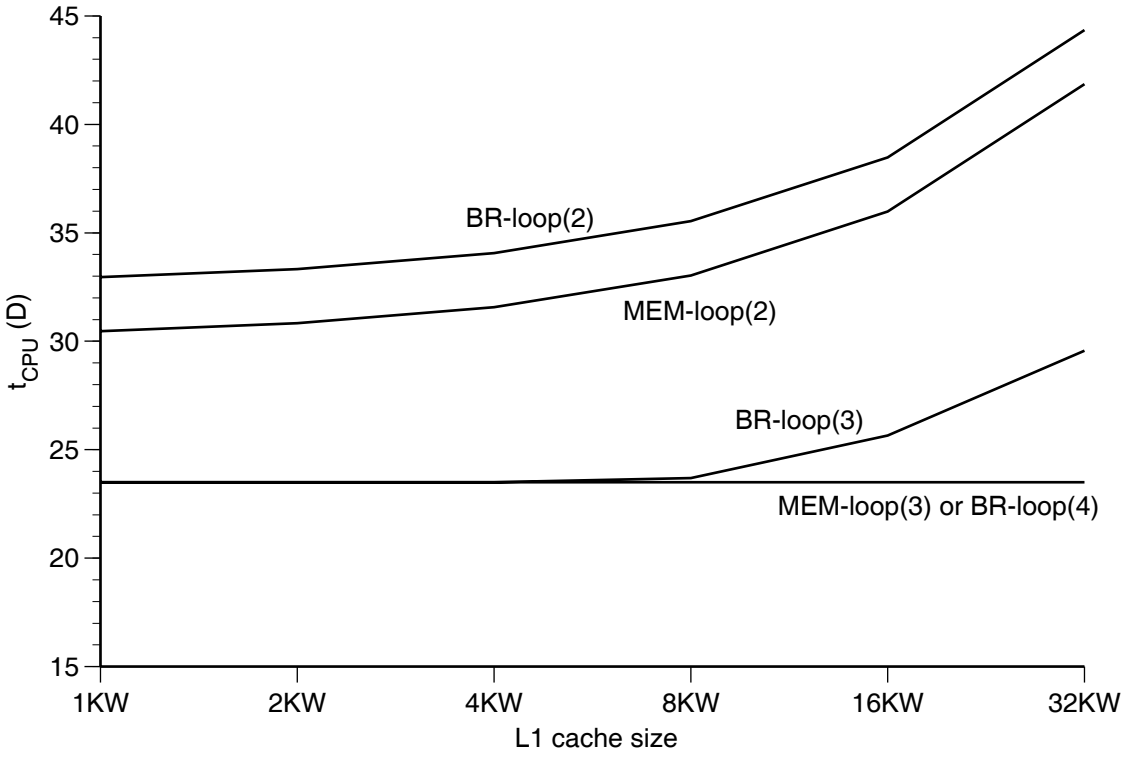

Figure 4.8: t_{CPU}versus cache size for the 4 KB SRAM chip.

cache sizes that vary from 1 KW to 32 KW. As expected, the cycle time monotonically increases with increasing cache size. The figure shows that for MEM loop and BR loop latency values of two, the BR loop is always critical and so sets the cycle time. Due to the two-cycle latency, t_{CPU} increases at a rate that is half of the rate at which t_{L1} increases. When the latency values of the MEM loop and BR loop are equal to three, for cache sizes smaller than 8 KW the SRAM access time t_{access} limits t_{CPU} to a value of $23.5D$. However, even if this were not the case, the EX loop delay would limit t_{CPU} to $23D$. For cache sizes larger than 8 KW, the BR loop again becomes the critical path and t_{CPU} increases at a third of the rate of t_{L1}. The value of t_{access} limits the t_{CPU} for all cache sizes when the latency values of the MEM loop and the BR loop are greater than or equal to four.

Figure 4.8 defines (4.3), the function $t_{\mathrm{CPU}}(\mathrm{L1}) = f(t_{\mathrm{L1}}, l_{\mathrm{L1}})$, for the L1 cache sizes and loop latency that will be used in the remainder of this thesis. The results presented in this figure do not include the effects of clock skew. However, using an H-tree clock distribution network it is possible to control clock skew to 30 ps or less

in a 15×15 cm area on an MCM [Bak90]. This amount of clock skew is less than 1 % of the cycle times we consider, therefore clock skew will have a very small effect on overall performance.

In summary, this section has developed the following formula:

$$t_{\mathrm{L1}} = (1 + 1.5B)D + \frac{4S_{\mathrm{L1}}}{B}\left[Z_0 C_{\mathrm{bond}} + 2\left(\overline{d}_{15}\sqrt{2}^{(B-15)}\right)^2 R_{\mathrm{MCM}} C_{\mathrm{MCM}}\right] + 2k_0$$

which can be used to predict the access time of MCM based caches in terms of normalized gate delay D. This formula was used to investigate the speed-size design space for SRAMs. It was shown that with MCM packaging technology, for the same sized cache, larger SRAMs do not necessarily result in lower cache access times. In fact, when the caches are integrated with the rest of the CPU datapath, the optimal clock schedule favors smaller SRAMs with lower access times. With these considerations coupled with the addition of yield and power arguments, a 4 KB SRAM was selected as the best SRAM chip size of the L1 cache. Finally, the CPU cycle time that results when this SRAM chip is used for various sizes of L1 cache was presented.

4.3 CPI of the L1 Cache

In this section the effect that the L1 cache organization has on CPI will be investigated by considering the L1-I and L1-D caches separately. Once the optimal L1-I and L1-D caches have been determined they will be combined into one coherent architecture. To facilitate the separation of the L1 cache into instruction and data parts it will be assumed that the write-buffer has read conflict checking hardware. This will allow L1 cache misses to fetch data from the L2 cache immediately, without waiting for the write-buffer to empty. Making this change lessens the impact of the L1 miss ratio on the performance of the write-buffer. It also isolates the effect on CPI of those factors that are directly related to the organization of the L1 cache. The effect that write-policy and write-buffer organization has on performance will be fully investigated in Chapter 5. Even though the L1-I and L1-D are separate, misses on either one will stall the entire CPU until the data is fetched from the L2 cache or from main memory.

Equation (4.5) shows that the CPI is a function of three factors that are determined by the L1 organization: L1 cache miss ratio, t_{CPU}, and L1 cache read latency. The effect of these factors will be determined using trace-driven simulation.

4.3.1 CPI of the L1-I Cache

As long as the CPU is executing instructions sequentially, the latency of the L1-I cache access has no effect on CPI. However, when a control transfer instruction (CTI) is executed, the next instruction to be fetched must be delayed until its address is known. Assuming that the branch condition of a conditional branch CTI can be evaluated in one cycle, the next instruction is delayed by $l_{\mathrm{Br}} - 1$ cycles, where l_{Br} is the latency of the BR loop. These cycles are called *branch delay slots*. Since most applications dynamically execute control transfer instructions 10–20 % of the time, a unit increase in $l_{\mathrm{L1-I}}$ will result in a 10–20 % increase in CPI. In order to reduce this effect on CPI a number of hardware and software schemes have been devised to reduce the effect of branch delay slots [Smi81, LS84, MH86, Lil88, HCC89, KT91]. Two representative schemes are evaluated here: hardware-based branch target buffering, and software-based delayed branching with optional squashing.

A branch target buffer (BTB) is a cache whose entries contain address tags and target addresses of branch instructions. Every instruction address is checked against the BTB's address tags. Those addresses that hit are predicted to be branches. The BTB simulated here uses a 2 b (bit) prediction scheme that is described in [LS84]. If the BTB predicts a branch taken, the target address is used as the next instruction address. If the BTB could be accessed in a single cycle and its predictions were always correct, it would completely hide the effect of branch delay slots. The size of the BTB that will be evaluated in this study is 256 entries. With two 32 b addresses plus 2 b of prediction per entry, the BTB requires approximately 2 KB of SRAM storage. This size represents the upper limit of an SRAM that could be placed on the GaAs CPU chip. Furthermore, larger BTBs will have access times that are too long to allow single cycle access, assuming that the EX loop sets the cycle time's lower bound.

In order to use delayed branches with optional squashing to reduce the ef-

fect of delay slots, the compiler must fill the delay slots of each branch instruction with useful instructions that are taken from among the instructions before the branch instruction, after the branch instruction, or from the branch target instructions. Instructions taken before the branch can always be executed no matter what direction the branch takes. In contrast, instructions from after the branch must be squashed if the branch is taken and instructions from the branch target must be squashed if the branch is not taken. Squashing requires dynamically converting the instructions to `noop` instructions. There is usually a code expansion associated with this approach because in order to schedule branch delay slots with instructions from the branch target the instructions must be replicated.

Evaluating the performance of these schemes for reducing the impact of branch delay slots requires knowledge of the number of delay slots, the number of these that are filled, the effect on CPI of the reduced static code size with the BTB, and the effect of the expanded static code size with delayed branches. To provide this information, a post-processor for MIPS object code was developed for mapping instruction addresses from those of the MIPS code (which assumes one branch delay slot) to those of a target architecture having a different number of delay slots. Using the information contained in this file during cache simulation, instruction addresses are translated from the MIPS I object code to the addresses of the object code of the architecture that is being simulated.

The post processor can create translation files to simulate ISAs with any number of delay slots. A file for an ISA without any delay slots is used for the BTB experiments. Such a file can be produced by, first of all, removing all `noop` instructions that appear after control transfer instructions (CTI) to create code without delay slots. Following this, the mapping between the basic block entry points of the original object code and the transformed code is recorded in the translation file. The code for a single delay slot ISA is the same as that produced by the MIPS compiler. However, the MIPS ISA does not include optional squashing and so it cannot fill the delay slot with instructions from after the branch or from the branch target. Cases where the compiler was unable to find an instruction from before the branch have a `noop`

instruction following the branch. This case is covered in the general (two or more) delay slot insertion scheme described below.

In the procedure for general delay slot insertion, b is the number of branch delay slots in the ISA that is to be simulated and r is the number branch delay slots that are filled from before the branch and s is the number of delay slots that must be filled from the sequential path or from the branch target path ($b = r + s$).

1. Check to see if the MIPS compiler inserted a `noop` instruction after the CTI. If it did, the CTI cannot be moved any higher in the basic block, Therefore, set $r = 0, s = b$ and go to step 3.

2. Move the CTI up in the basic block as far as allowed by the data dependencies of the instructions that are above the branch. Besides the CTI, no attempt is made to rearrange other instructions in the basic block. Set r and s.

3. Determine the prediction of the CTI. Backward branches and unconditional jumps are predicted to be *taken*. Forward branches are predicted to be *not taken*.

4. Insert s `noop` instructions after CTI that are predicted to be taken in order to simulate the instructions that are replicated from the target path. After doing this, set $s = 0$ for register-indirect jumps because the target of these jumps is not computable at compile time.

After this procedure is completed, associated with each CTI is the number, s, of instructions in the delay slots that may need to be squashed and a flag that indicates whether the CTI is predicted to be taken or not-taken. The value of s, the branch prediction along with the basic-block entry-point mappings are placed in the translation file.

The idea behind the translation files is to use a trace from a computer with an ISA that has c branch delay slots to simulate the instruction referencing behavior of a computer with b branch delay slots, where b is not equal to c. Cache simulation using the translation files works as follows. Each basic block entry-point instruction

address that comes from an instrumented benchmark is translated to a new address using the data in the translation file. This address is used to simulate l sequential instructions, where l is the length of the translated basic-block. For the purpose of cache simulation, we are not concerned with the specific mechanism used to squash instructions in the delay slots, we are only concerned that the correct instruction reference stream is produced. To get the correct reference stream, it is sufficient to check the prediction of each CTI. If the prediction of the CTI is that it will be taken and the prediction is correct, then the value of s associated with the CTI is added to the instruction address of the target basic block thus leaving $l - s$ instructions to be executed in the target basic block. This assumes that s instructions of the target basic block have been executed in the delay slots of the CTI. If there are less than s instructions in the target basic block then the delay slots are assumed to be padded with `noop` instructions. If the prediction is that the CTI will be taken and the prediction is incorrect, then the instruction address of the sequential basic block is left unchanged. Doing this simulates the effect of the extra delay slot instruction references. If the prediction is that the CTI will not be taken and the prediction is correct, then no action is taken because the sequential instructions are in the delay slots of the CTI. However, if the CTI is predicted not to be taken and the prediction is incorrect, then s extra instruction references are made in the sequential basic block before control is transferred to the target basic block.

Figure 4.9 shows, for 1 to 3 delay slots, the average static code size increase for all the benchmarks (referenced to that of an ISA having no delay slots). The increase arises from filling the delay slots of the 47 % of CTIs that are predicted to be taken. Measurements taken from our benchmarks indicate that the MIPS compiler is able to fill 54 % of all first branch delay slots with instructions from before the CTI. For CTIs predicted to be taken, this number decreases to 52 %. The rest of the delay slots for these CTIs must be filled with instructions from the branch target. The delay slots of register indirect jumps cannot be filled with target instructions and so are filled with `noop` instructions. Register indirect jumps make up roughly 10 % of the CTIs in our benchmarks.

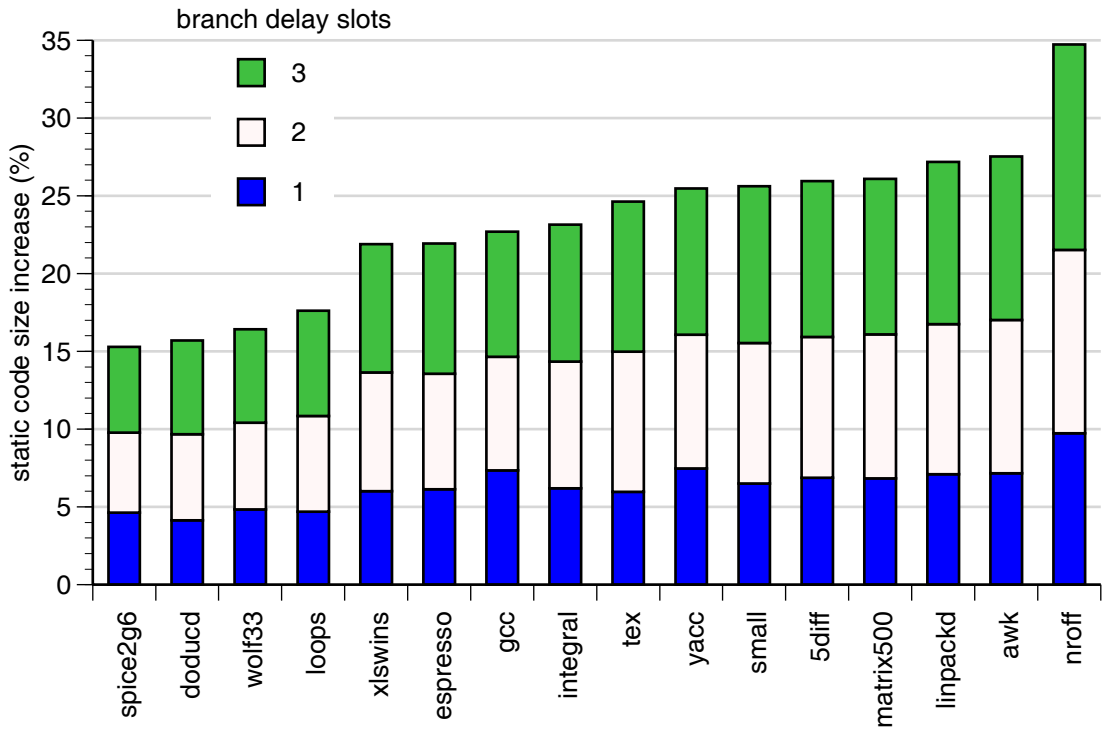

Figure 4.9: Static code size increase versus the number of branch delay slots.

The increase in CPI as the number of branch delay slots increases comes from two sources: (1) increases in the number of L1-I cache misses due to the larger code size and (2) the extra instruction references that are executed when the static branch prediction is incorrect. In addition, the two sources are related because more instruction references tend to create more L1-I cache misses. To measure the effect of the extra L1-I cache misses, various L1 cache configurations were simulated as the number of branch delay slots was varied from 0 to 3. Apart from delay slots, the other major parameters that were varied in these simulations were the L1-I cache size, S_{L1}, the line or block size, B_{L1}, and the L1 miss penalty, P_{L1}.

In general, the L1 miss penalty P_{L1} in CPU cycles for off-MCM L2 caches is calculated using the following formula:

$$P_{\mathrm{L1}} = \left\lceil \frac{(8\mathrm{ns} + \frac{B_{\mathrm{L1}} \times 16\mathrm{ns}}{W_{\mathrm{tr}}})}{t_{\mathrm{CPU}}} \right\rceil \tag{4.14}$$

where B_{L1} is the block or line size of the L1 cache and W_{tr} is the transfer width from the L2 cache to the L1 cache. This formula reflects that B_{L1} is an integer multiple

Delay slots	CTIs Predicted Taken % of total	% correct	CTIs Predicted Not-Taken % of total	% correct	Cycles per CTI	Additional CPI
1	47	93	53	49	1.092	0.012
2	47	93	53	49	1.339	0.044
3	47	93	53	49	1.67	0.087

Table 4.4: Performance of branch prediction versus number of branch delay slots. The numbers of predict-taken and predict-not- taken CTIs are expressed as percentages of the total number of CTIs that were executed. The CTIs made up 13 % of all instructions that were executed.

of W_{tr}. This formula models the fact that it takes 8 ns to drive the address off of the MCM and to receive the data back from the L2 cache SRAMs, and that the L2 cache cycle time is 16 ns.

Using (4.14), cache experiments were performed for L1-I cache sizes from 1 KW to 32 KW, for transfer widths from 1 W to 4 W, and for the branch delay slots from 1 to 3. The rest of the system configuration is the same as the base architecture with the changes that were described at the beginning of this section. The results of the L1-I cache experiments are shown in Figures 4.10–4.12. These figures reveal that the number of delay slots has a measurable impact on the performance of the L1-I cache. This impact diminishes as the cache size and transfer width are increased. For the smallest L1-I cache size of 1 KW (4 KB) the rate of increase in CPI is roughly 0.06 per delay slot for a transfer size of 1 W. When the transfer size is increased to 4 W the rate of CPI increase drops to 0.015 per delay slot. For the largest L1-I cache size of 32 KW, the rate of increase in CPI as the number of delay slots increases is quite small. The rate of increase varies from 0.014 to 0.004 CPI per delay slot as the transfer width is increased from 1 W to 4 W.

The block size is optimized for the lowest CPI for each value of transfer width used in the L1-I experiments. For transfer widths of 1 and 2 W the block size of 4 W is optimal. The optimal block size increases to 8 W when the transfer width is increased to 4 W. Even larger optimal block size would result if an interleaved L2 cache that had a higher transfer rate than one transfer per 16 ns.

Table 4.4 lists, for 1, 2 and 3 delay slots, the static branch prediction statis-

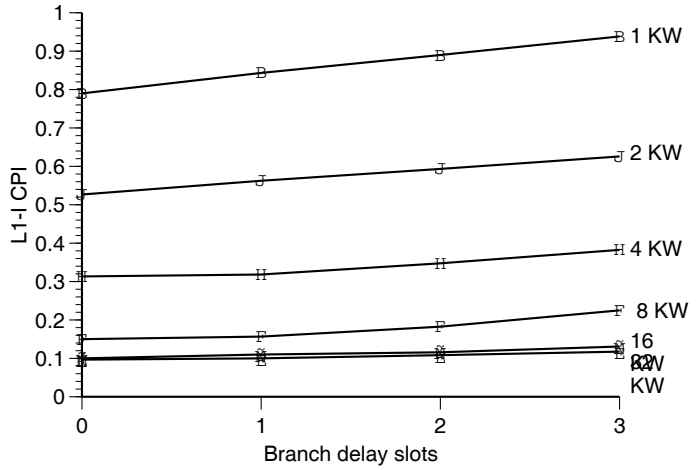

Figure 4.10: Effect of branch delay slots on L1-I performance: $W_{\mathrm{tr}} = 1\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

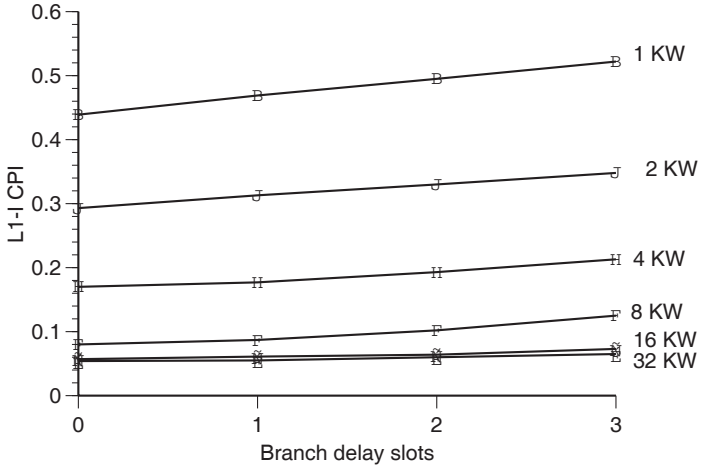

Figure 4.11: Effect of branch delay slots on L1-I performance: $W_{\mathrm{tr}} = 2\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

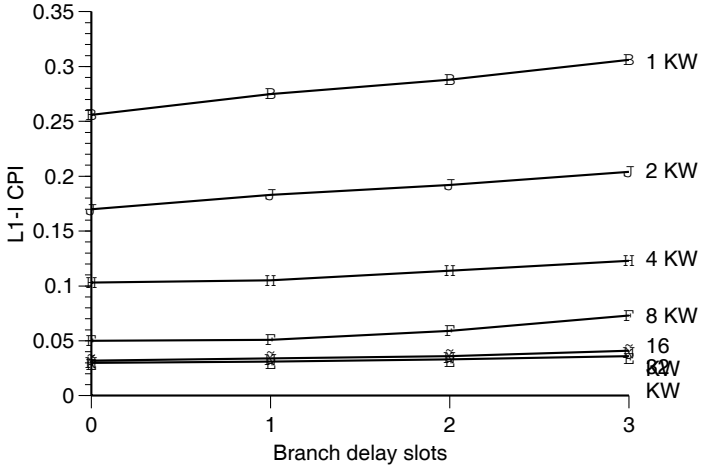

Figure 4.12: Effect of branch delay slots on L1-I performance: $W_{\mathrm{tr}} = 4\,\mathrm{W}$, $B_{\mathrm{L1}} = 8\,\mathrm{W}$.

Benchmark	BTB hit	incorrect prediction	BTB miss taken	BTB miss not taken
5diff	99.44	13.76	0.39	0.17
awk	86.83	16.35	9.94	3.22
doducd	90.10	10.53	7.36	2.55
espresso	95.90	21.75	2.56	1.54
gcc	74.77	16.80	16.66	8.57
integral	99.13	16.24	0.63	0.23
linpackd	99.60	12.05	0.31	0.09
LFK12	100.00	9.50	0.00	0.00
matrix500	99.99	1.06	0.01	0.00
nroff	88.20	4.03	8.08	3.72
small	99.70	19.59	0.30	0.00
spice2g6	85.48	17.87	9.27	5.26
tex	82.19	13.45	13.15	4.66
wolf33	76.69	25.01	14.56	8.75
xlswins	80.02	10.92	15.59	4.39
yacc	99.21	10.78	0.52	0.27
Harmonic mean	91.46	15.21	5.63	2.91

Table 4.5: A summary of the 256 entry BTB performance. The numbers are expressed as percentages of the total number of CTIs.

tics, cycles per branch, and additional CPI due to extra instruction reference cycles. The data in this table support two conclusions. First, static branch prediction with optional squashing is an effective scheme for mitigating the branch-delay penalty. For example, since 13 % of instructions executed are CTIs, three branch delay slots could increase CPI by 39 %; in fact, the increase is only 8.7 %, a reduction of nearly 80 % due to good branch prediction. Secondly, the effect of instruction cache misses should not be ignored when considering the performance of aggressive static branch prediction schemes. Though this is less important for large caches and small miss penalties, it is noteworthy that the increase in CPI due to increased cache misses with three delay slots is 9 % for a 1 K-W L1-I cache with a 10-cycle miss penalty (see Figure 4.11).

The performance of the 256 entry BTB is summarized in Table 4.5 for all benchmarks. The data shows that although the BTB achieves a hit rate of over 91 %, incorrect predictions reduce its overall branch prediction accuracy to 86%. This is still

Delay slots	Cycles per CTI	Extra CPI
1	1.44	0.057
2	1.65	0.082
3	1.85	0.11

Table 4.6: BTB prediction performance.

better than the static prediction accuracy of 70%. However, the overall effectiveness of the BTB method is reduced because an extra cycle is required to update the BTB with the correct information every time there is a BTB miss or an incorrect prediction. When these cycles are included in the branch penalty, the performance of the BTB is reduced to that shown in Table 4.6.

A comparison of Tables 4.4 and 4.6 shows that the static scheme performs better. This is because static rearrangement allows 0.5 to 0.8 of the delay slots to be filled with instructions from before the CTI, so that fewer cycles are wasted even if the CTI prediction is incorrect. The BTB scheme loses one cycle per delay slot every time a CTI misses the BTB or the CTI prediction is incorrect. One could argue that the relatively small size of the BTB compromises its performance; recall, though, that the BTB was restricted to 256 entries to ensure single cycle access, which is necessary to make this scheme worthwhile. Though they did not investigate the effects of code expansion, other researchers have shown that static branch prediction techniques using sophisticated program profiling and fetch strategies are competitive with much larger BTBs [HCC89, KT91]. Of course, for static prediction, the additional CPI due to increased L1-I misses (see Table L1-I-2wpt) must be considered; for small cache sizes and large miss penalties, this would give the performance edge to the BTB approach. Nevertheless, because its performance is roughly comparable and its hardware cost is lower, the static prediction scheme is used in the remainder of the L1-I cache experiments.

Figures 4.13–4.15 plot the total CPI for the same values of L1-I cache size, numbers of delay slots, and transfer widths that were used in Figures 4.10–4.12 Two

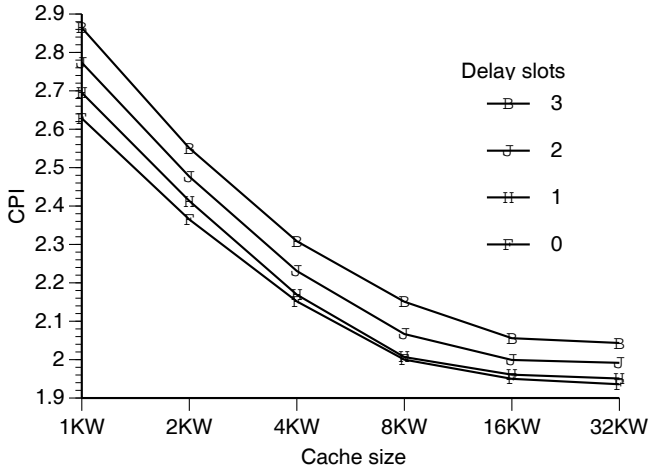

Figure 4.13: Branch delay slots versus L1-I cache size: $W_{\mathrm{tr}} = 1\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

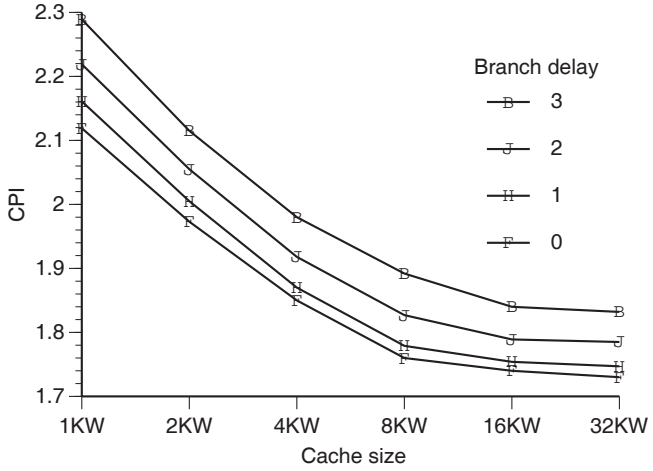

Figure 4.14: Branch delay slots versus L1-I cache size: $W_{\mathrm{tr}} = 2\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

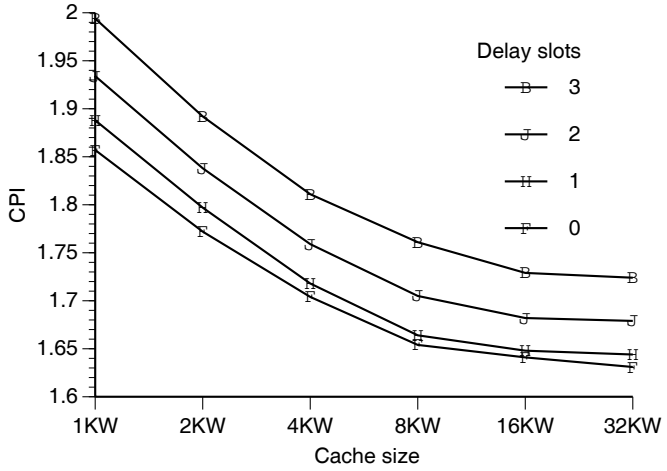

Figure 4.15: Branch delay slots versus L1-I cache size: $W_{\mathrm{tr}} = 4\,\mathrm{W}$, $B_{\mathrm{L1}} = 8\,\mathrm{W}$.

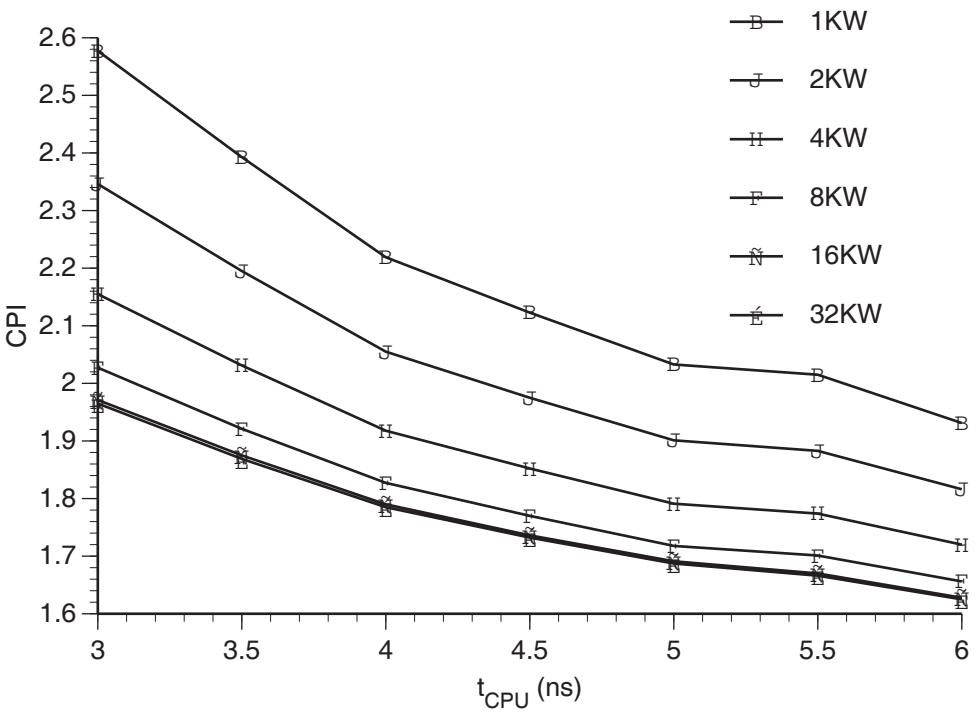

Figure 4.16: t_{CPU} versus L1-I cache size: $b = 2$, $W_{\mathrm{tr}} = 4\,\mathrm{W}$, $B_{\mathrm{L1}} = 2\,\mathrm{W}$. The curves are not smooth because of the ceiling function in (4.14)

observations can be made from these figures. First, as expected, doubling the transfer width dramatically decreases CPI because it reduces the miss penalty of both L1-I and L1-D. The effect of a wide transfer width is most pronounced for the smallest size cache of 1 KW. For this size an increase in transfer width from 1 W to 4 W reduces the CPI by 1.0. For the largest L1-I cache size of 32 KW, the reduction in CPI for the same increase in transfer width is 0.3. Second, is that for all transfer widths that were considered, it is always possible to decrease the CPI of the system by doubling the cache size and increasing the number of delay slots by one, for L1-I cache sizes of 1–16 KW. The reason for this is that in this region of L1-I cache size the relative increase in CPI from increasing the number of delay slots (0.03–0.15) is less than the decrease in CPI from doubling the cache size (0.05–0.2).

The function h of (4.5) which relates CPI to the organization of the L1 cache also depends on t_{CPU}. This dependence is illustrated in Figure 4.16 which plots the t_{CPU} versus cache size tradeoff for a system having two branch delay slots and a transfer width of 4 W. If the number of delay slots is changed, smaller caches will be

affected more by the code size increase. Likewise, smaller caches are affected more by changes in transfer width because they have higher miss ratios. Figure 4.16 shows that CPI decreases as t_{CPU} increases because the miss penalty of the L1 cache in cycles defined in (4.14) decreases as t_{CPU} increases. Furthermore, the miss penalty of the L2 cache, *i.e.* the main memory access time, expressed in CPU cycles decreases as t_{CPU} increases.

4.3.2 CPI of the L1-D Cache

The L1-D cache supplies data to the CPU when *load* instructions are executed. The number of CPU cycles, between the execution of a load instruction and the time at which the data arrives at the CPU is determined by $l_{\mathrm{Mem}} - 1$, where, l_{Mem} is the latency of the MEM loop. These cycles are called load delay slots.

The MIPS ISA provides only one memory addressing mode for all load instructions. This mode, called *register plus displacement*, uses a 16 b signed displacement from a 32 b general purpose register (GPR). To aid our discussion of load delay slots, the following fragment of code, including the `lw` instruction is given as an example:

```
subu r5, r5, r4
       •
       •
       •
lw r3, 100(r5)
       •
       •
       •
addu r4, r3, r2
```

The `lw` instruction loads a word from memory location 100 plus the contents of address register `r5` into `r3`. It is preceded by an instruction that subtracts the contents of `r4` from `r5` and places the contents in `r5`. It is followed by instruction that adds the contents of `r3` to `r2` and places their sum into `r4`. We define c as the number of instructions between the last instruction to modify the address register and the load instruction, d to be the number of instructions between the load instruction and the first instruction that uses the result of the load, e to be the sum of c and d, and l to

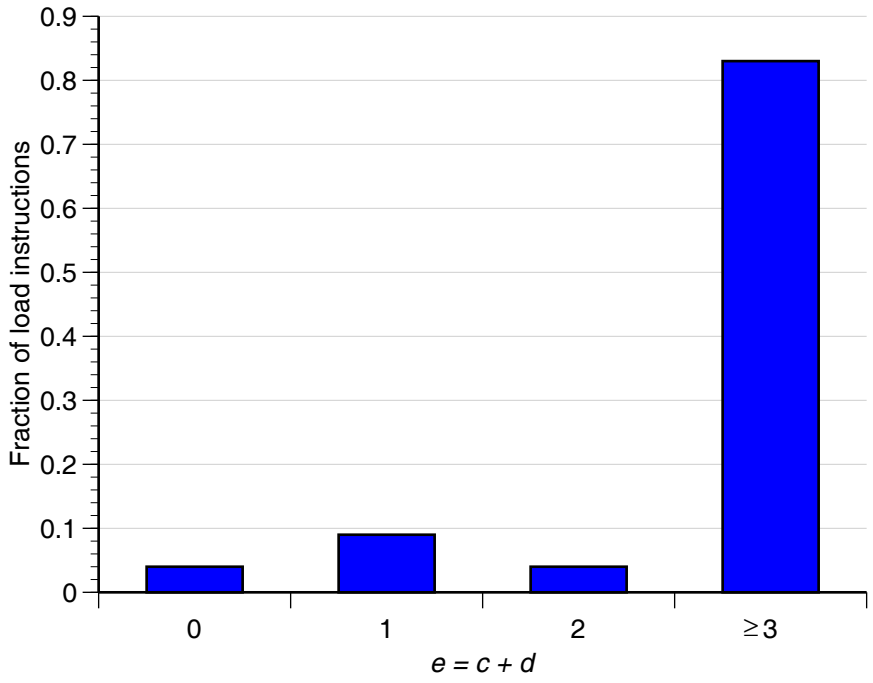

Figure 4.17: Histogram of e values for benchmark suite used in this study.

be the number of load delay slots in the ISA. In this example, it is possible to execute the `lw` directly after the `subu` instruction, implying $c \geq 0$, but the `add` instruction cannot be executed until l cycles after the `lw` instruction.

If nothing is done to hide the lost cycles due to load delay slots, the CPI will increase by the fraction of instructions that are loads, times the number of load delay slots in the ISA. In the benchmark suite used in our experiments this fraction is 0.25. To reduce the effect of load delay slots, load instructions can be moved up in the program, away from the ALU instructions that depend on them, so that $d \geq l$. This may be done statically at compile time or dynamically at execution time. The restriction that a load instructions must execute after the instruction that modifies its address register limits the number of delay slots that can be hidden to e.

The number of delay slots that can be hidden is determined from a distribution of e values, such as Figure 4.17. The large fraction (over 80 %) of loads that have e values of three or more demonstrates great opportunity for moving load instructions away from where their results are used. The reason for this high percentage of loads having $e \geq 3$ is that most variable references are to global static variables or

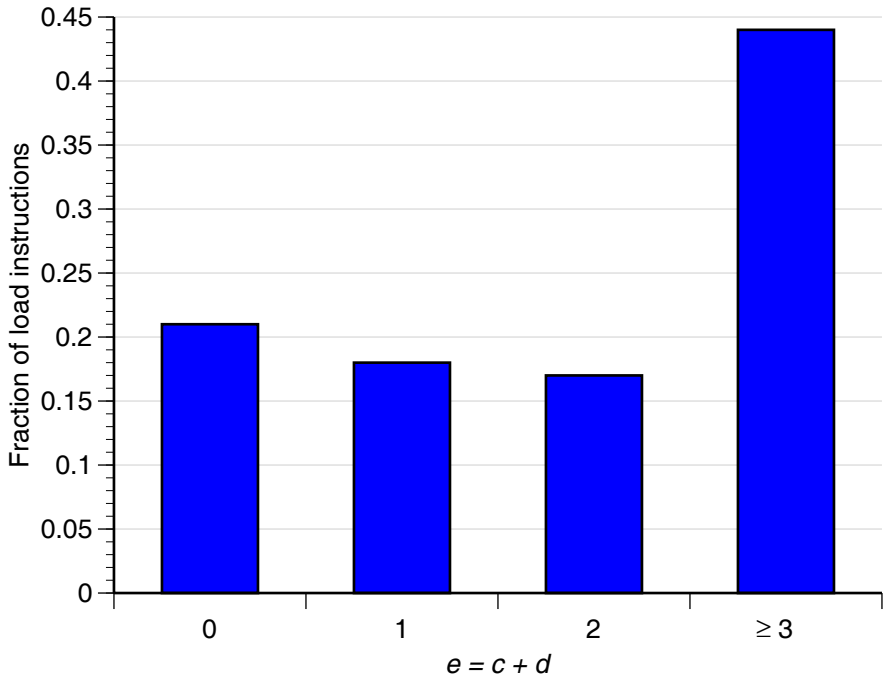

Figure 4.18: Static address register distance versus load delay slots.

to local automatic variables whose address registers do not change often. Program measurements reveal that over 90% of array and structure references are to global variables and over 80% of scalar references are to local variables [PS82]. The MIPS compiler allocates most global static variables from a 64 KB area of memory called the *gp area*. Variables in this area are addressed using a single dedicated register that is set once at the beginning of the program. Local automatic variables are addressed using the stack pointer register, which only changes at procedure call entry points. Furthermore, the MIPS compiler attempts to address as many memory locations as possible from the same address register [CCH+87].

Though, Figure 4.17 would indicate that most load delay slots can be hidden dynamically for an ISA with $l \leq 3$, the presence of control transfer instructions reduces the number of opportunities for hiding load delay slots at compile time. Figure 4.18 illustrates this by presenting the data of Figure 4.17 with the restrictions introduced by basic block boundaries. These boundaries prevent load instructions from being moved away from the instructions that use them, causing the value of c to be reduced. A comparison of Figures 4.17 and 4.18 shows clearly that these boundaries change

Delay slots	Static Delay slots per load	Static CPI	Dynamic Delay slots per load	Dynamic CPI
1	0.21	0.05	0.04	0.01
2	0.62	0.18	0.19	0.05
3	1.21	0.29	0.39	0.08

Table 4.7: The increase in CPI due to load delay slots.

the distribution of e, so that far fewer delay slots can be hidden in an ISA having $0 < l \leq 3$.

Based on the data in Figures 4.17 and 4.18, Table 4.7 shows the CPI increase that results from 1, 2 and 3 load delay slots. The data in this table is calculated using a dynamic frequency for load instructions of 0.25. Even though the data in Table 4.7 shows that dynamic load delay slot hiding could potentially be much better at hiding load delay slots than static instruction scheduling, dynamic schemes would require out-of-order instruction execution, extra register-file ports, and a separate load address adder. This extra hardware will increase the cycle time. Rather than trying to estimate the change in t_{CPU}, we assume static instruction scheduling in the remainder of this analysis, and refer to Table 4.7 to estimate the performance of dynamic scheduling or to estimate how much the cycle time could be increased in a dynamic scheme before it reaches the performance crossover point with static instruction scheduling.

Figures 4.19–4.21 show CPI versus L1-D cache size for 0 through 3 delay slots and for transfer widths of 1 W and 4 W. As in the L1-I experiments the block sizes of the caches have been optimized for the refill latency and transfer rate. Each figure shows the effect that load delay slot values, varying between zero and three, have on CPI. We assume that load instructions are interlocked. This avoids the code expansion associated with the need to insert `noop` instructions into load delay slots which cannot be filled with useful instructions. Figure 4.20 shows that the effect of load delay slots becomes more significant for values greater than one. In fact, in order to decrease CPI after increasing the number of load delay slots from one to two

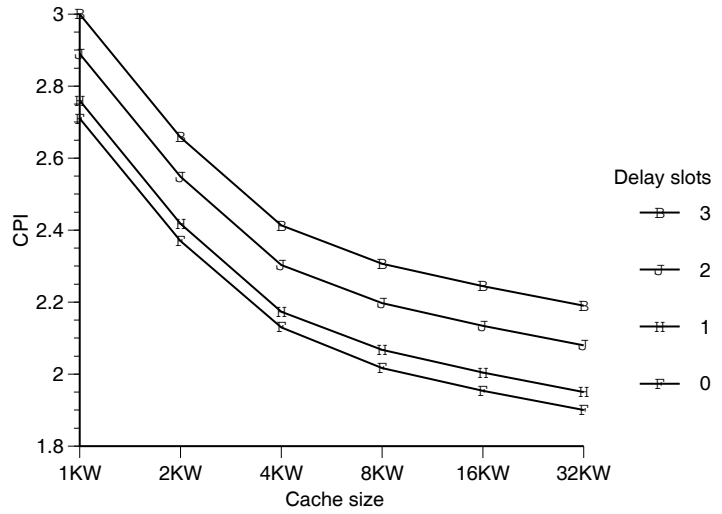

Figure 4.19: Load delay slots versus L1-D cache size: $W_{\mathrm{tr}} = 1\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

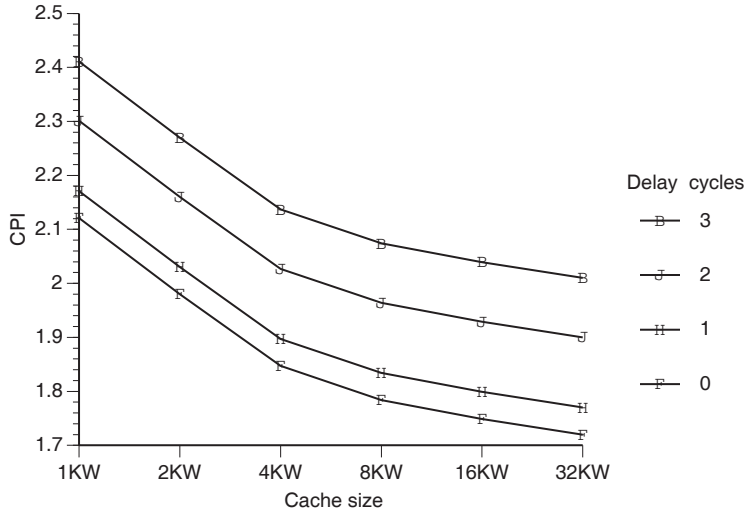

Figure 4.20: Load delay slots versus L1-D cache size: $W_{\mathrm{tr}} = 2\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

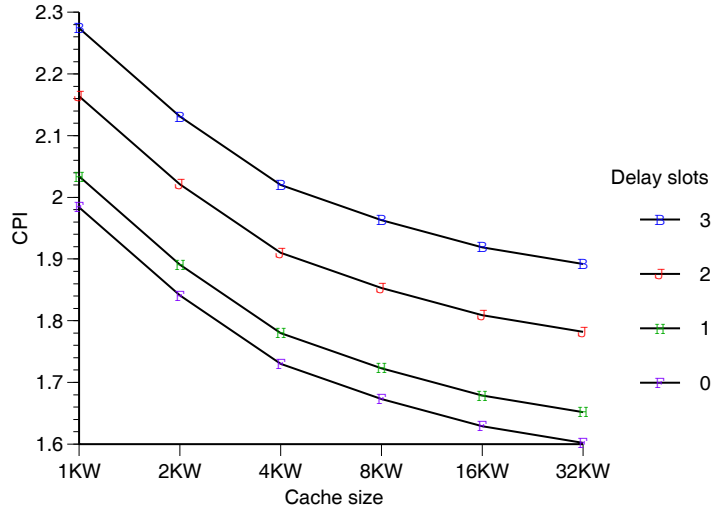

Figure 4.21: Load delay slots versus L1-D cache size: $W_{\mathrm{tr}} = 4\,\mathrm{W}$, $B_{\mathrm{L1}} = 8\,\mathrm{W}$.

	L1-I		L1-D	
Cache size	MPI	$1 - \mathrm{RM}$	MPI	$1 - \mathrm{RM}$
1 KW	0.026		0.036	
2 KW	0.017	0.33	0.025	0.30
4 KW	0.010	0.30	0.015	0.42
8 KW	0.005	0.48	0.010	0.33
16 KW	0.004	0.25	0.007	0.28
32 KW	0.003	0.25	0.005	0.27

Table 4.8: Time independent metrics for the L1 cache. The number of misses is expressed by the misses per instruction (MPI) ratio. The ratio of misses of one cache size and a cache of twice the size is denoted by RM. The value $1 - \mathrm{RM}$ indicates the fraction of misses that were eliminated.

requires a four-fold increase in cache size, and this can only be done if the original cache is 2 KW or smaller.

In comparisons with results of the L1-I experiments shown in Figures 4.13–4.15, Figures 4.19–4.21 show that the L1-D cache has a wider variation in CPI over the same range of cache sizes. In the case of the 4 W transfer width and zero delay slots the range of CPI for L1-I is 0.25 while the range of CPI for L1-D is 0.4. There are two reasons for this. The first is that instruction references have higher temporal and spatial locality than data references and thus create fewer cache misses. The second reason is that changing the L1-D cache size affects the number of writes in the write buffer. Although, the write buffer never fills up (recall that L1-D uses a write back policy) an L1 cache miss must wait for a write in progress to complete before fetching data from the secondary cache. Larger L1-D caches decrease the frequency of these waits and so improve the performance of the machine by more than the decrease in miss ratio. The performance of write buffers will be fully investigated in the next chapter. However, the fact that write buffer performance affects the overall performance of the system implies that ignoring it will lead to inaccurate performance predictions.

Table 4.8 presents the time independent performance metrics for L1-I and L1-D caches with a block size of 8 W. The poorer locality of data accesses is shown by the increased values of misses per instruction (MPI). Even though data accesses

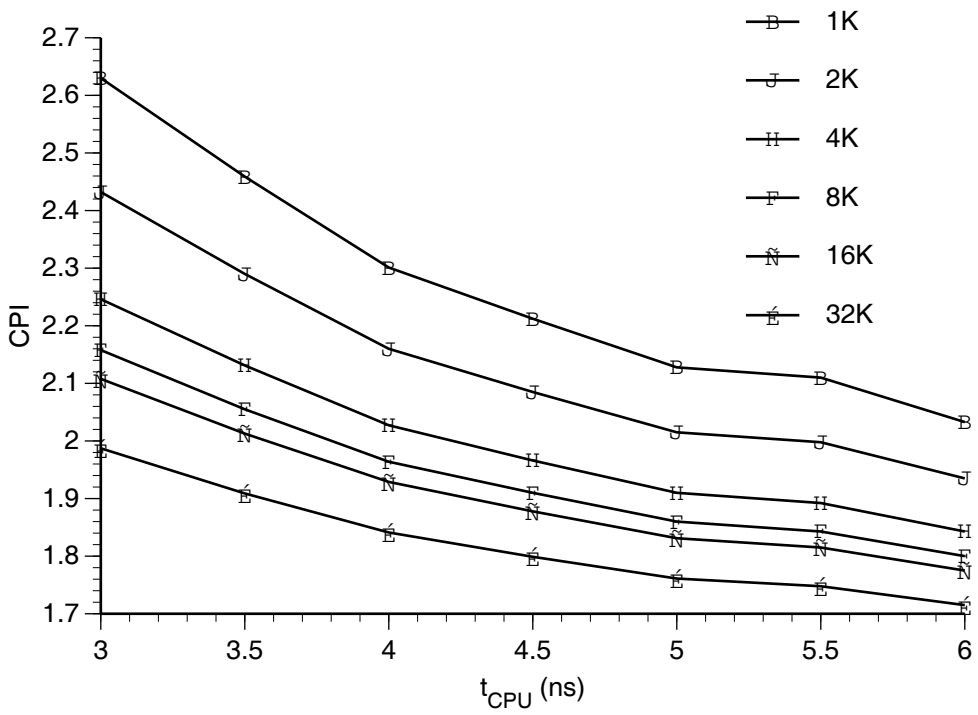

Figure 4.22: t_{CPU} versus L1-D cache size: $l = 2$, $W_{\mathrm{tr}} = 2\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

have poorer locality, doubling the cache is equally effective at reducing misses. The reduction in the number of misses is between 30 % and 40 % for each doubling of cache size of both L1 caches. This is consistent with the data reported by other researchers [Sto90].

Figure 4.22 is the result of plotting CPI versus t_{CPU} for various L1-D cache sizes. The parameters used in this figure are two load delay slots and a transfer width of 2 W. Varying the number of delay slots will shift the plot up or down without changing the shape of the plot. Changing the transfer width will affect the smaller caches more because they have larger values of MPI. Figure 4.22 shows that increasing t_{CPU} lowers CPI which gives the illusion of increased performance if CPI is used as the performance metric. The figure also shows larger caches have lower values of CPI, which might also indicate that they provide higher levels of performance. However, the next section will show that when TPI is used as the performance metric different conclusions will result.

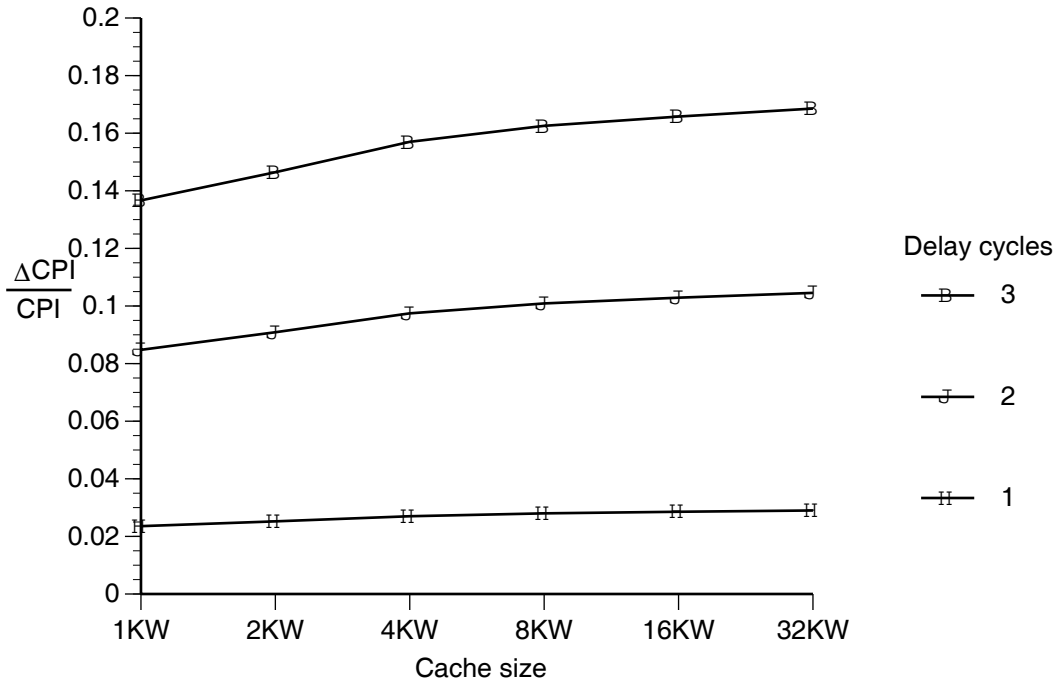

Figure 4.23: $\frac{\Delta \text{CPI}}{\text{CPI}}$ for L1-D.

4.4 L1 TPI

In this section the results of the L1-I and L1-D CPI experiments are combined with the t_{CPU} data to predict the performance of L1 caches.

Pipelining cache access has the potential to decrease t_{CPU}, but in order to increase system performance, the relative decrease in t_{CPU}, $\frac{\Delta t_{\text{CPU}}}{t_{\text{CPU}}}$, must be greater than the relative increase in CPI, $\frac{\Delta \text{CPI}}{\text{CPI}}$. In Figure 4.23 $\frac{\Delta \text{CPI}}{\text{CPI}}$ is plotted for L1-D caches in order to assess the t_{CPU} change required to improve performance. The relative CPI is measured against the CPI of an ISA with no load delay slots. This figure shows that as the number of delay slots is increased, the relative decrease in t_{CPU} required to improve performance grows larger. For 2 delay slots, the decrease is less than 10 %. This suggests that performance will be improved if t_{CPU} can be reduced by more that 10 %. The figure also shows that as the cache size grows, the required improvement in t_{CPU} grows larger, suggesting that pipelining the cache access path is less effective for larger caches. In general, this result indicates that if the CPI of a system is low, deep pipelining will require significant reductions in t_{CPU} before the performance of

L1 size	BR-loop(1)	BR-loop(2)	BR-loop(3)	BR-loop(4)	MEM-loop(1)	MEM-loop(2)	MEM-loop(3)	MEM-loop(4)
1KW	10.0	5.0	3.5	3.5	9.2	4.6	3.5	3.5
2KW	10.1	5.1	3.5	3.5	9.4	4.7	3.5	3.5
4KW	10.3	5.2	3.5	3.5	9.6	4.8	3.5	3.5
8KW	10.9	5.4	3.6	3.5	10.1	5.1	3.5	3.5
16KW	12.0	6.0	4.0	3.5	11.2	5.6	3.7	3.5
32KW	14.2	7.1	4.7	3.5	13.4	6.7	4.5	3.5

Table 4.9: Optimal cycle times for L1 caches for $B = 4$. All times are in nanoseconds. The numbers in parentheses specify the latency of the loop.

L1 size	BR-loop(1)	BR-loop(2)	BR-loop(3)	BR-loop(4)	MEM-loop(1)	MEM-loop(2)	MEM-loop(3)	MEM-loop(4)
1KW	10.0	5.0	3.5	3.5	9.2	4.6	3.5	3.5
2KW	10.1	5.1	3.5	3.5	9.4	4.7	3.5	3.5
4KW	10.3	5.2	3.5	3.5	9.6	4.8	3.5	3.5
8KW	10.8	5.4	3.6	3.5	10.0	5.0	3.5	3.5
16KW	11.8	5.9	3.9	3.5	11.0	5.5	3.7	3.5
32KW	13.7	6.9	4.6	3.5	13.0	6.5	4.3	3.5

Table 4.10: Optimal cycle times for L1 caches for $B = 8$. All times are in nanoseconds. The numbers in parentheses specify the latency of the loop.

the system increases.

To determine exactly how t_{CPU} varies with cache size and pipeline depth we used $minT_C$ to estimate t_{CPU} for L1 cache sizes of 1 to 32 KW. Tables 4.9 and 4.10 tabulate the output of $minT_C$ for block sizes of 4 W and 8 W. In these tables the latency of the BR loop and MEM loop is varied between 1 and 4. The timing analysis of each loop was performed with circuit structures that include the loop and the EX loop. This was done to ensure that only the EX loop or the loop of interest would be on the critical path. The data in Tables 4.9– 4.10 differs from the data shown in Figure 4.8 because the extra tag SRAMs have been added to the delay calculations.

The data in Table 4.9 for a block size of 4 W shows that for a latency value of one the BR loop and MEM loop limit t_{CPU} to greater than 10 ns. Since, a latency of one corresponds to zero delay slots, this data clearly demonstrates that requiring the

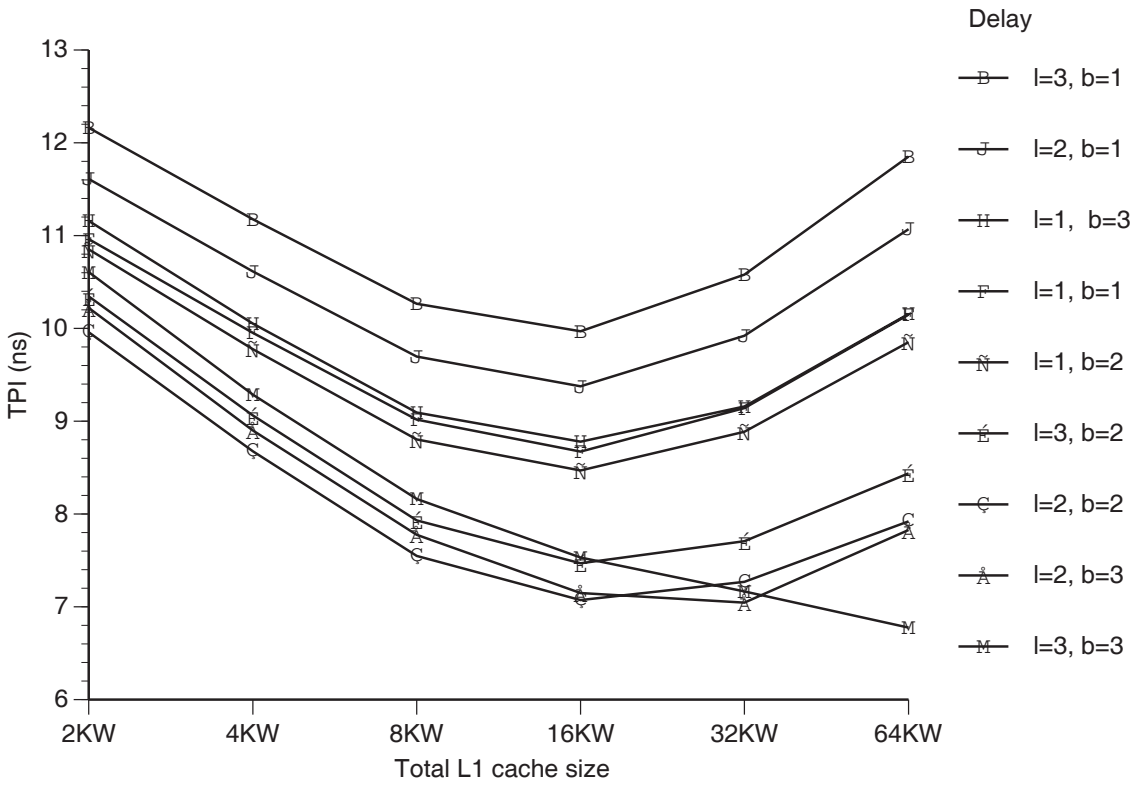

Figure 4.24: Cache size versus number of delay slots: $W_{\text{tr}} = 2\,\text{W}$, $B_{\text{L1}} = 4\,\text{W}$.

L1 cache to be accessed in the same cycle as the execution unit will lead to excessively long cycle times compared to the ALU add time which is 2.1 ns in this case. When the latency values of the BR loop and MEM loop are increased to 4 the EX loop is critical for all cache sizes and the cycle time is limited to 3.5 ns.

The data in Table 4.10 shows that when the block size is increased to 8 W the minimum cycle times of the CPU will be decreased for cache sizes larger than 8 KW and for loop latency values less than 4. The reason for this is that half as many tag SRAMs are required for these caches which reduces the MCM delay.

So far the L1 cache experiments have been presented in terms of L1-I or L1-D. In order to combine a particular L1-I organization and an L1-D organization we take the maximum t_{CPU} of each as the new system cycle time t_{CPU}. The result of doing this is shown in Figure 4.26.

A number of conclusions can be drawn from Figure 4.24. It shows that when the L1 cache is divided equally between L1-I and L1-D, performance is maximized

when the number of branch delay slots is equal to the number of load delay slots, *i.e.*, $b = l$. This follows from the fact that pipelining the different sides of the L1 cache to different depths causes the t_{CPU} set by one side to be shorter than that of the other. Since the side with the longest cycle time will set the system cycle time, the extra pipelining on the other side will be wasted, *i.e.*, there will be extra CPI without the benefit of reducing t_{CPU}. Figure 4.24 also shows that for every combination of load and branch delay slots, there is an L1 cache size that maximizes performance. Maximum performance is reached for medium size caches at TPI = 6.8ns, when $b = 3$, $l = 3$, $S_{\mathrm{L1}} = 64\,\mathrm{KW}$, and $t_{\mathrm{CPU}} = 3.5\,\mathrm{ns}$. This observation brings us to the final observation, namely, that increasing the number of branch and load delay slots is able to increase performance because doing so reduces the dependence of t_{CPU} on the size of the cache. This allows larger caches to be accessed without increasing cycle time.

If dynamic out-of-order load execution were used instead of static load delay scheduling, a new maximum performance of TPI = 6.2ns could be reached when the number of branch and load delay slots are both equal to three ($l = 3, b = 3$) and the combined L1 cache size is 64 KW. The value of TPI is strongly dependent on the cycle time. We calculate that if the implementation of the out-of-order load execution required more than a 10 % increase in t_{CPU} the performance of the dynamic scheme would be worse than the performance of the best organization with static load delay scheduling.

Different L1 penalties change the performance and location of the optimal design points: higher penalties increase both the L1 cache size and pipeline depth and lowers overall performance as shown in Figure 4.25. Lower penalties have the opposite effect, as shown in Figure 4.26, where maximum performance is reached at TPI = 6.61ns ($b = 2$, $l = 2$, $S_{\mathrm{L1}} = 16\,\mathrm{KW}$, and $t_{\mathrm{CPU}} = 3.5\,\mathrm{ns}$). Smaller refill penalties also make it possible to take advantage of the fact that increasing the number of branch delay slots increases CPI less than a comparable increase in load delay slots. This can be done by using a larger size L1-I cache than L1-D cache and pipelining the access of the L1-I cache more deeply. When this is done, as shown in Figure 4.27. , the maximum performance is reached at TPI = 6.5ns ($S_{\mathrm{L1-I}} = 32\,\mathrm{KW}$

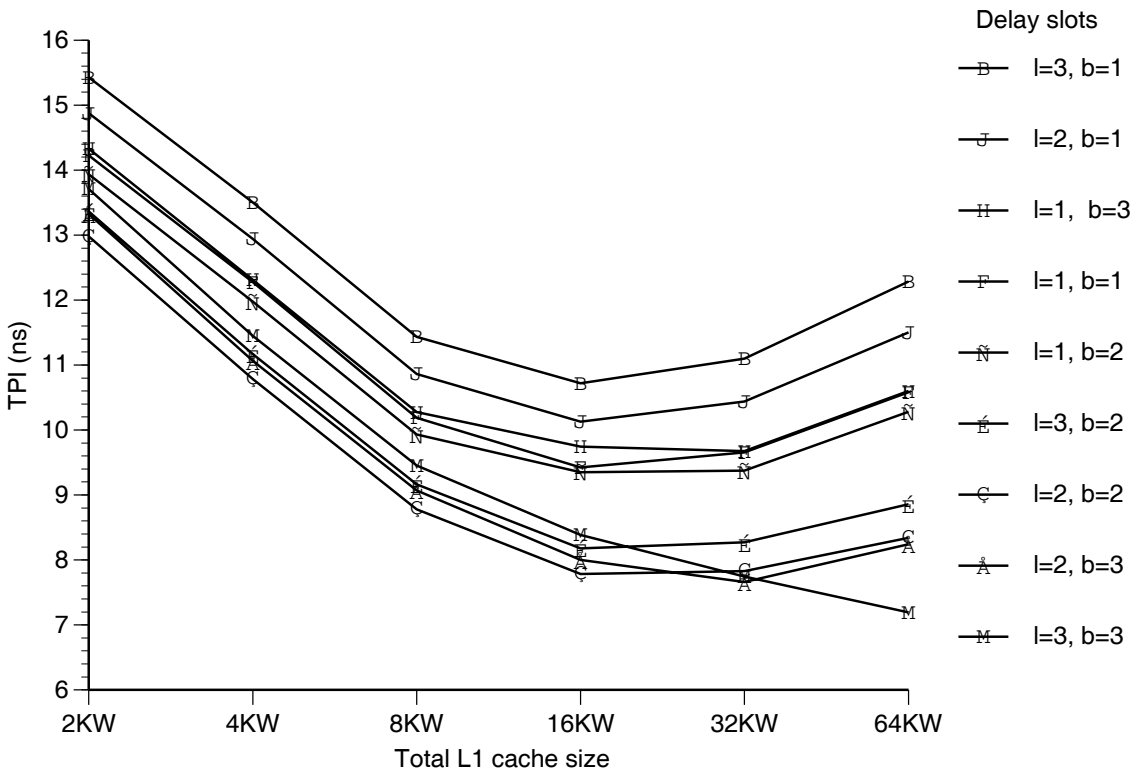

Figure 4.25: Cache size versus number of delay slots: $W_{\mathrm{tr}} = 1\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

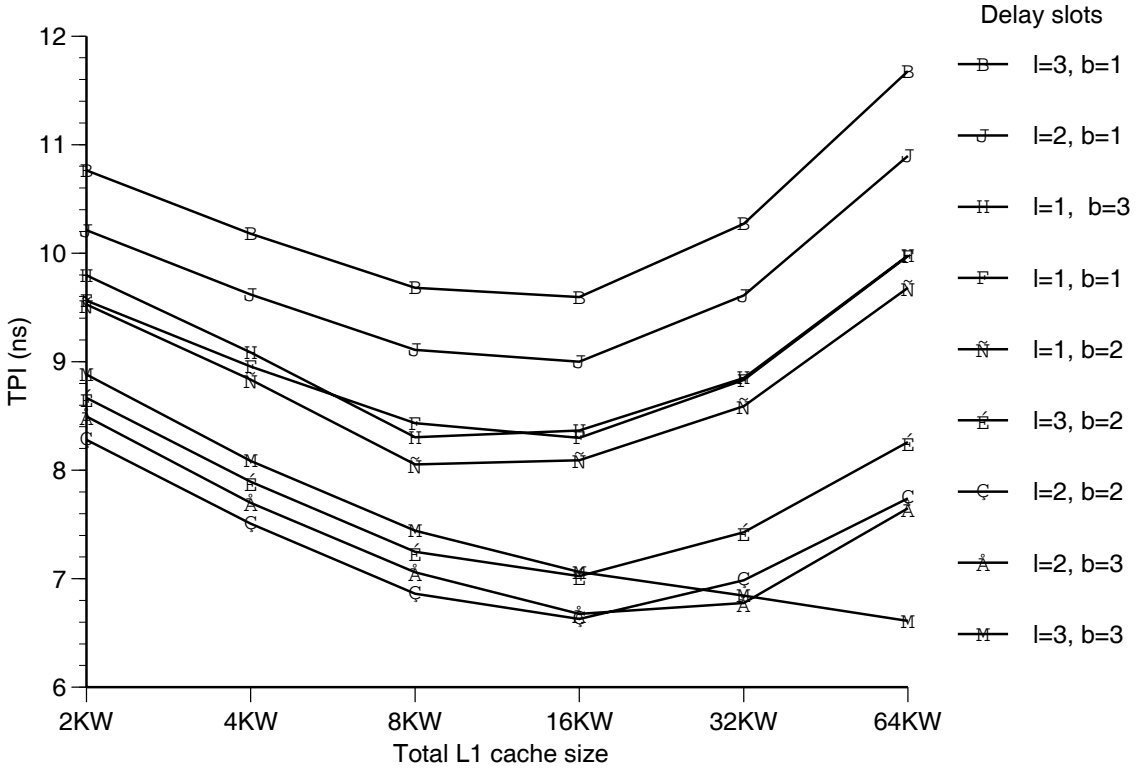

Figure 4.26: Cache size versus number of delay slots for $B_{\mathrm{L1}} = 4\,\mathrm{W}$ and $P_{\mathrm{L1}} = 6\,\mathrm{cycles}$.

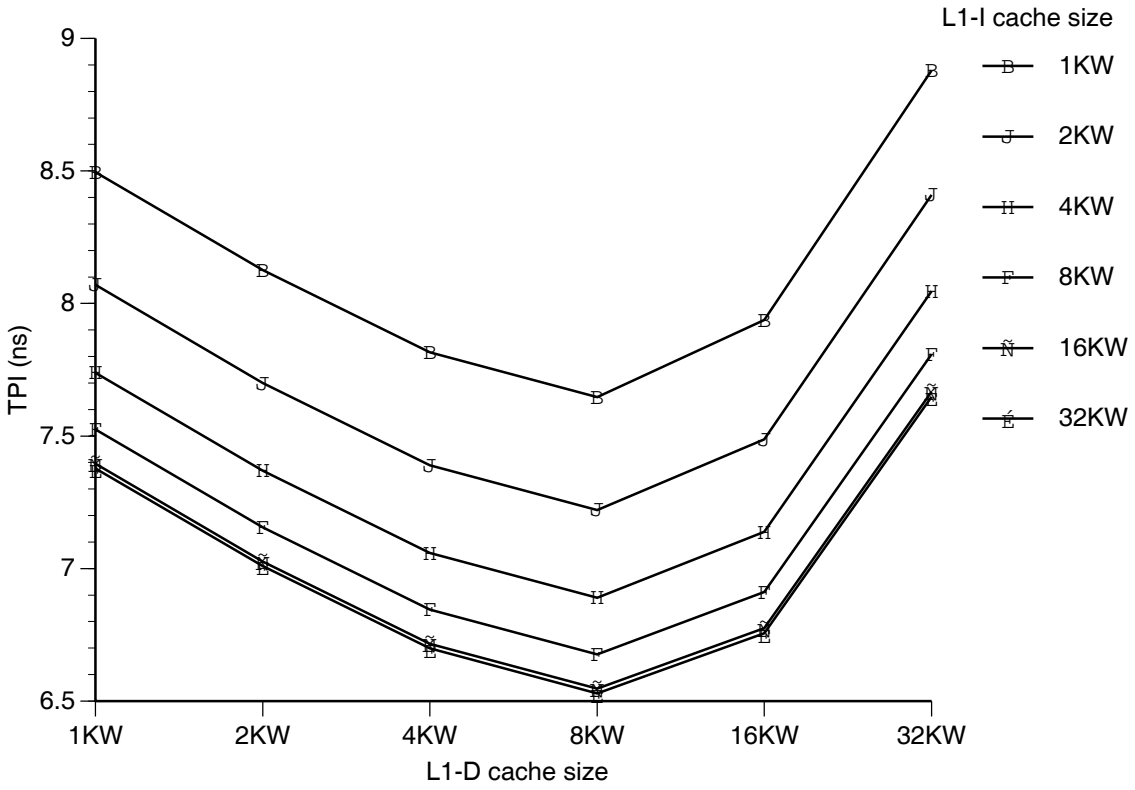

Figure 4.27: L1-I cache size versus L1-D cache size: $b = 3$, $l = 2$, $W_{\mathrm{tr}} = 4\,\mathrm{W}$, $B_{\mathrm{L1}} = 4\,\mathrm{W}$.

and $S_{\mathrm{L1-D}} = 8\,\mathrm{KW}$.) This performance is the maximum that any of the L1 cache organizations attained. The reason for the differences in the organizations L1-I and L1-D are due to longer BR loop delay compared to the MEM loop and the lower branch delay slot CPI penalty of increasing the number of delay slots compared to the load delay slot.

4.5 Conclusions

In this chapter the multilevel design optimization procedure has been applied to the design of an L1 cache. Doing this has revealed a number of general principles for on-chip and on-MCM L1 cache design.

Temporal analysis of on-chip and MCM-based caches showed that the access time of on-chip caches increases logarithmically with increases cache size and that the access time of MCM-based caches increases linearly with increases in cache size. The expression for on-chip cache access time is technology independent, while the expression for MCM based caches uses the electrical parameters of the MCM and chip bonding method. The expressions can be used to determine how the access time varies with cache size for on-chip caches and cache size and SRAM chip size for MCM based caches.

Analysis of base architecture showed that for reasonable caches sizes, the L1 caches will determine the cycle time. Furthermore, unless access to the L1 caches is pipelined, t_{CPU} will be up to a factor of five longer than the time it takes to perform integer addition. To further characterize the design space. $MinT_C$ was used to investigate the tradeoff among t_{CPU}, L1 cache sizes, SRAM sizes and degree of pipelining. The results showed that in the case of MCM based caches, larger SRAM chips with longer access times do not necessarily provide the lowest overall cache access time. Furthermore, in a pipelined cache environment, trading less SRAM access time for more MCM delay to equalize the delays between the latches can lead to lower values of t_{CPU}. Trace-driven simulation was used to simulate a range of L1-I and L1-D cache sizes, block sizes, and L1 miss penalties. Using novel trace-driven

simulation methods, the effect of the number of branch and load delay slots has on CPI was evaluated. The results of the branch-delay experiments showed that delayed branching with optional squashing can provide better performance than a BTB that is small enough to allow single cycle access. The extra cache misses caused by the larger code size of the statically-scheduled scheme causes a small increase in CPI that is comparable to the increase in CPI caused by the branch delay slots. The effect of the load-delay slots cannot be hidden as well as branch delay slots with static instruction scheduling.

The individual results of this chapter were combined into one graph using TPI as the performance metric. This graph reveals that there are significant performance benefits to deeply pipelining the L1 cache for a CPU with a peak execution rate of one instruction per cycle. The conclusion is that adding latency in terms of cycles can improve overall performance because the effect of these extra cycles can be hidden. Further, more latency can be tolerated on the L1-I side than the L1-D side because the extra cycles created by branch delay slots are easier to hide. This suggests that for maximum performance the L1-I should be larger and pipelined more deeply than L1-D. Finally, this chapter shows that the benefits of adding latency to the cache access path are two fold. First, it reduces the t_{CPU} and the dependence of t_{CPU} on cache access time. Second, it allows larger caches to be accessed without increasing t_{CPU}, thus lowering CPI. These results suggest that the cache size versus set- associativity tradeoff may need to be re-examined. If t_{CPU} is less dependent on the access time of pipelined L1 caches, then increasing the associativity of the cache to lower the miss ratio will have a larger performance benefit for pipelined caches.

CHAPTER 5

A TWO-LEVEL CACHE FOR A GAAS MICROPROCESSOR

In the last two chapters we have argued that besides pipelined L1 caches, L2 tags should be placed on the CPU to speed up L2 writes. The performance of a processor could be increased greatly if the entire L2 cache could be placed on the MCM, however, MCM area and power constraints prevent this. Therefore, it is necessary to determine exactly how to get the maximum performance from a limited MCM area. In this chapter the trade-offs between technology and organization will be investigated for MCM technology and two-level cache organizations. To help with this, the knowledge of L1 cache design trade-offs that was gained from the last chapter will be used together with new insights that are gained from a study of other aspects of the cache organization. The specific organizational issues that will be addressed are the L1 cache write policy, the organization of the L2 cache and the concurrency in the cache-hierarchy. These issues will be considered in the context of a GaAs CPU that is implemented using MCM technology.

5.1 L2 organization

The effects of secondary cache size and organization on performance are investigated by looking at four candidate cache organizations: unified direct-mapped, unified two-way associative, split direct-mapped and split two-way associative. To model the multiplexing and comparison logic necessary to implement two-way associative caches the access time of the these caches is increased from 6 to 7 cycles, or

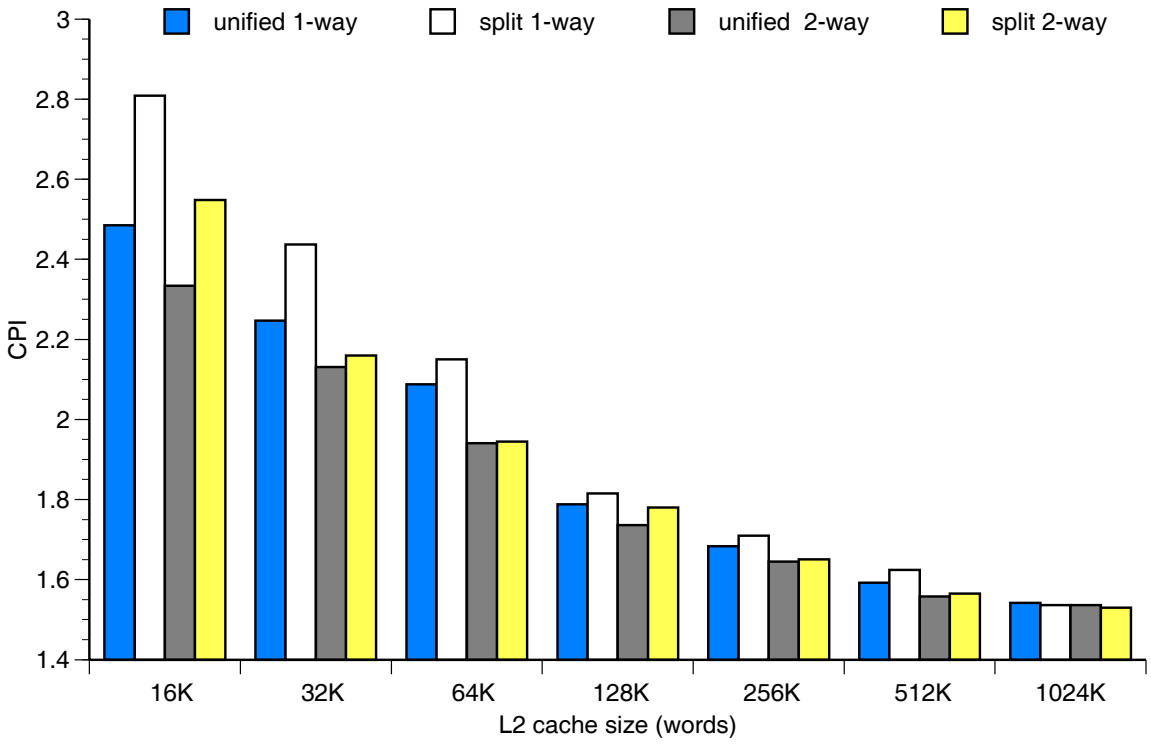

Figure 5.1: Performance of various L2 sizes and organizations. Direct-mapped caches have a 6 cycle access time; 2-way set-associative caches have a 7 cycle access time.

by 16.7 %. A split cache is logically partitioned into instructions and data. A split cache can be implemented by using the high-order bit of the cache index to interleave between the instruction and data halves of the cache. Doing this to a direct-mapped cache does not increase the access time or require more cache access bandwidth.

Figure 5.1 shows the performance of these four cache organizations. This figure shows that for a direct-mapped and two-way associative cache, splitting does almost as well as a unified cache for cache sizes of 64 KW to 512 KW. For cache size of 1024 KW, performance is improved.

Two processes access the secondary cache: instruction fetching and data accessing. These two processes never share address space in main memory, but in the cache they can interfere with one another because of mapping conflicts (two memory locations mapped to the same cache block). Mapping conflicts are more significant in direct mapped caches than caches with higher degrees of set associativity, and contribute more to performance degradation.

size (words)	unified 1-way	split 1-way	unified 2-way	split 2-way
16K	8.54	10.91	6.85	7.40
32K	6.41	8.19	5.14	5.56
64K	4.91	4.93	3.45	3.76
128K	2.63	2.94	1.89	2.31
256K	1.60	1.99	1.01	1.20
512K	1.14	1.30	0.70	0.83
1024K	0.67	0.61	0.56	0.54

Table 5.1: L2 miss ratios for the sizes and organizations of Figure 5.1. The miss ratios are expressed as the number of misses per 1000 instructions.

Cache misses can be classified into compulsory, conflict and capacity misses [Hil87]. In a fully-associative cache, all misses are classified as either compulsory or capacity misses. In a set-associative cache the number of misses minus the number of misses of a fully-associative cache of the same size are classified as conflict misses. The results in Figure 5.1 show that the conflict misses between instruction and data references can be reduced without affecting access time by dividing the cache into separate instruction and data portions, provided the cache size is large (greater than 1 MW) as shown in Table 5.1. For smaller caches the capacity misses of the data accesses of a cache that is half the size are greater than the instruction and data conflict misses. As the cache size grows, both conflict and capacity misses decrease, but conflict misses decrease at a slower rate for each doubling in cache size [HP90]. Splitting the cache can reduce the miss ratio for large caches, because as the cache size increases, conflict misses comprise a larger component of the miss ratio.

To further describe the effect that splitting the L2 cache has on performance the speed-size trade-off curves for the L2 instruction cache (L2-I) and L2 data cache (L2-D) are shown in Figures 5.2 and 5.3. The effect of writes on L2-D is ignored in order to simplify the comparison between L2-I and L2-D. The data in the figures are the result of varying the speed and size of the L2-I and L2-D caches from the base architecture. Both sets of curves show the same trend: the marginal performance increase due to increasing cache size is smaller for larger cache sizes. However, the exact shape and value of the L2-I and L2-D speed-size trade-off curves is quite dif-

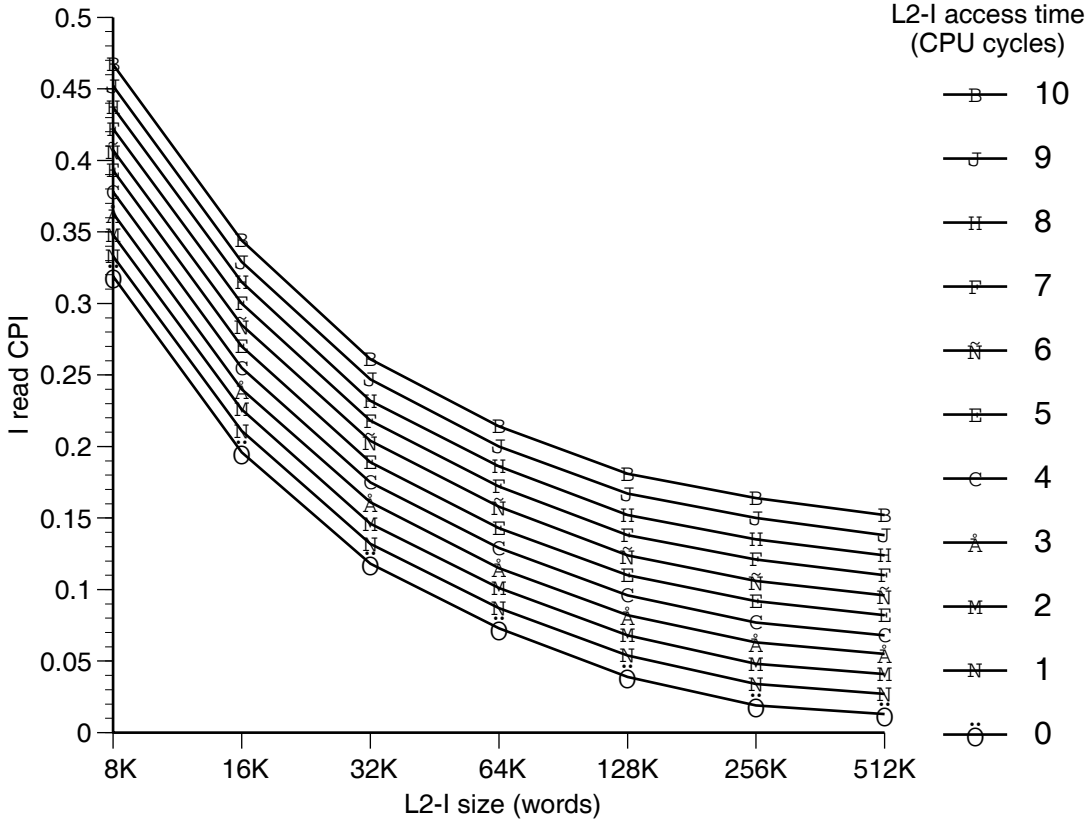

Figure 5.2: The L2-I speed-size trade-off with a 4KW L1-I.

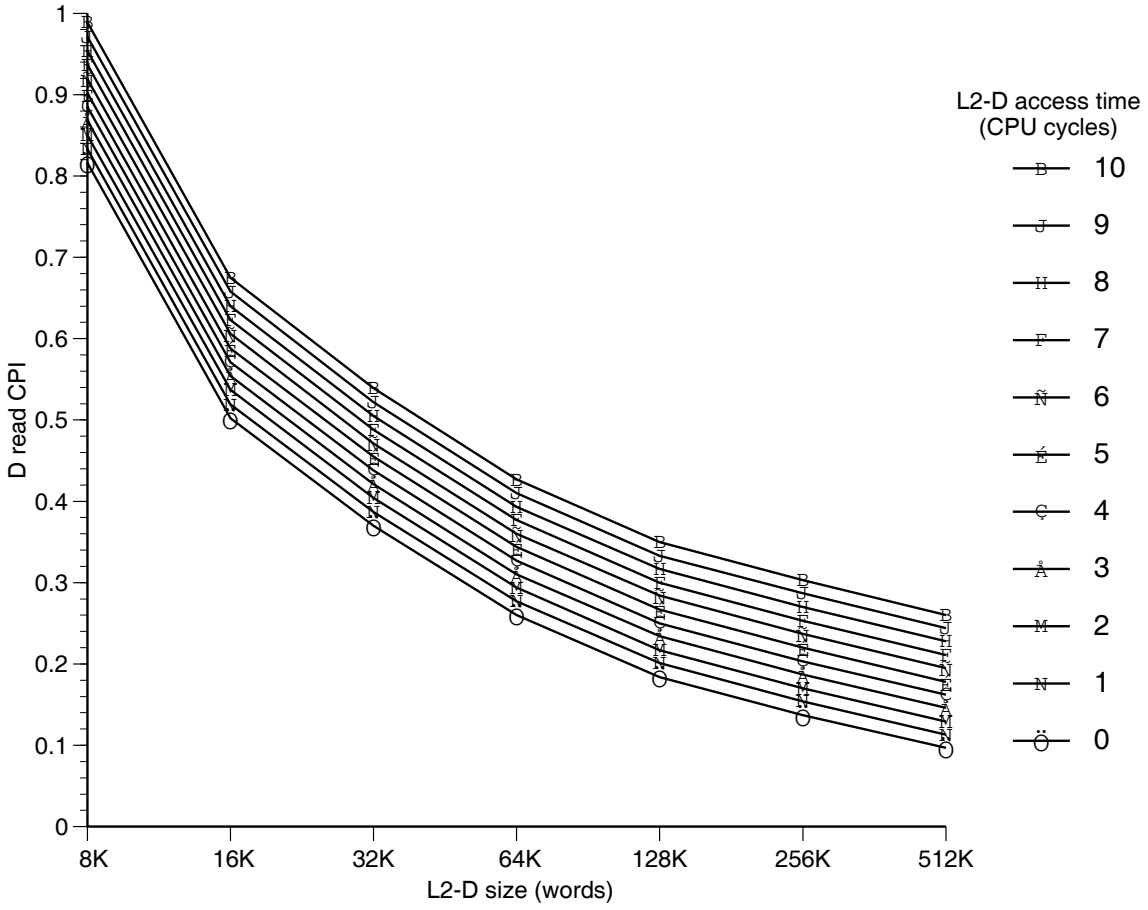

Figure 5.3: The L2-D speed-size trade-off with a 4KW L1-D.

ferent. The L2-I cache CPI degradation varies from 0.46 to 0.02 and the curves are flatten out for cache sizes greater than 128 KW. For cache sizes larger than 128 KW the change in CPI is less than 0.01 per doubling in cache size. In contrast, L2-D the cache CPI degradation varies from 1.0 to 0.1 and it is still decreasing at the rate of 0.06 CPI per doubling in cache size at a cache size of 512 KW. This data suggests that the optimum data cache size is roughly 8 times as large as the optimum instruction cache. Two-way associative split L2 caches show the same trends, although the curves are shifted downward due to the lower miss rates and the L2-D cache performance improves more than the L2-I cache.

Thus we see that the speed-size trade-offs for secondary instruction and data parts of a split secondary cache are quite different. To take advantage of this fact, the L2-I cache should be implemented in a faster technology than the L2-D cache even if the speed is bought at the cost of density. To see how to do this for the base architecture, consider the curve for the zero cycle L2-I cache in Figure 5.2. For a cache size of 32 KW the performance of the zero cycle cache is better than a 256 KW 6-cycle access cache. The reason for this is that with a zero cycle L2-I cache L1-I misses are free. Although it is impossible to construct a zero cycle cache, it is possible to change the two-level instruction cache into a single level 32 KW. This can be done using the cache pipelining techniques that were discussed in Chapter 4. Figure 5.4 shows the effect of branch delay slots on a single level I-cache with a 32 W block size. This figures shows that the block size of 32 W is effective at reducing the extra misses caused by increasing the number of branch delay slots. For example, the total increase in CPI of increasing the number of branch delay slots in a 32 KW cache is 0.029.

Figures 5.2, 5.4 and the results of Chapter 5 indicate that performance can be increased by using a single level 32 KW I-cache with a branch delay of three instead of a two level I-cache (4 KW L1-I) with a branch delay of one. To match the pipelining of the I-cache the number of load delay slots must be increased to two. However, Figure 5.3 clearly shows that it is not profitable to trade speed for size in the case of L2-D. Therefore, the two level organization of the L2-D cache should not

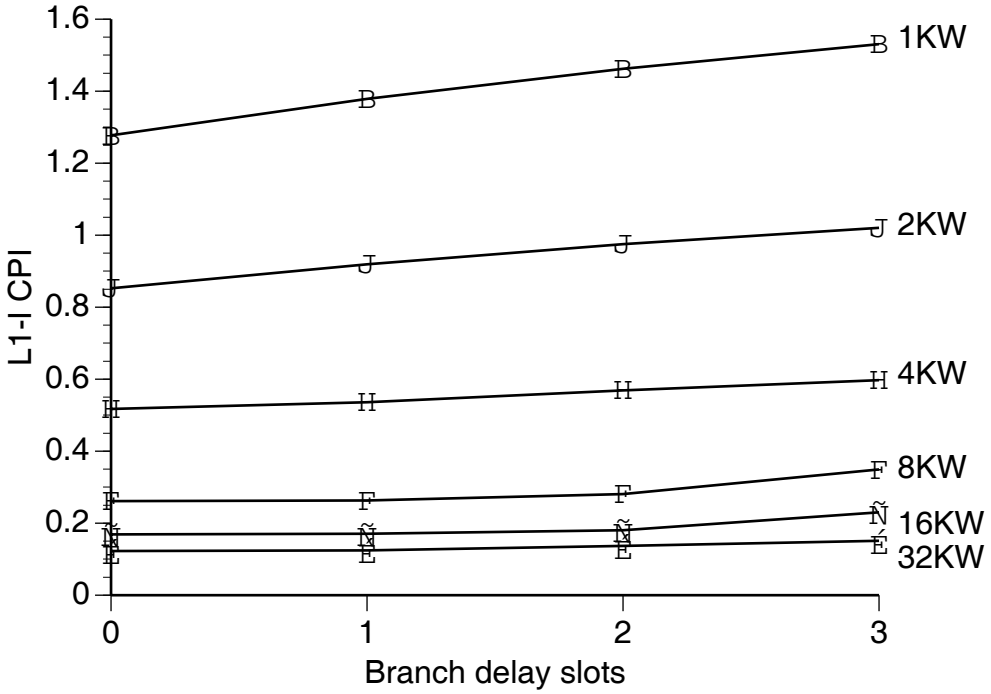

Figure 5.4: The effect of branch delay slots on a single level I-cache.

be changed.

A 32 KW cache with a 32 W block size will require 33 1 KW SRAMS. This number of chips together with all others on the MCM is at the outer limit of packing and power density. It is also at the limit for forced air cooling. To reduce the area and cooling requirements the size of the I-cache can be reduced to 16 KW. If this is done, CPI of the I-cache will increase by almost 0.1 from 0.15 to 0.25. However, the CPI can be brought back down again if the I-cache can be made 2-way set associative. This reduces the CPI of the I-cache down to 0.12 which is slightly better than a 32 KW cache.

Two-way set associative caches have lower miss ratios and hence lower values of CPI loss but they also have greater access times [Hil88]. However, a pipelined cache lowers or eliminates the dependence of t_{CPU} on the cache access time and so makes set-associativity more attractive. With the GaAs and MCM technology that has been proposed in this thesis, a 16 KW cache that is part of a BR loop with a latency of four (three branch delay slots) is not on the critical path that sets t_{CPU}. In fact there are more than 2 ns to spare before the Br loop becomes critical. In this technology, 2 ns

is enough time to implement set-associativity, especially if the I-cache tags are placed in the cache management unit (CMU) chip. The total amount of memory required for the tags is 26 Kb[1]. This assumes that the L1-D cache has an 8 W block size, that the I-cache uses virtual tags and that the L1-D cache uses physical tags. Placing the caches onto the CMU would also reduce the number of pins on the CMU chip by 40, further reducing the power consumption of the chip.

Reducing the size of the I-cache by half and making it 2-way set-associative reduces power, cooling requirements and MCM area and increases access time, but still results in increased performance. This performance improving trade-off between size and speed is made possible by pipelining the cache. Once 2-way set associativity has been implemented there is little reason not to make the cache 4-way set associative because this only requires two more comparators inside the CMU and two extra SRAM output enable lines. The benefits of 4-way associativity are that the I-cache CPI loss is reduced for 0.12 to 0.11 and that the size of the tag memory is reduced from 26 Kb to 20 Kb. The size of tag memory will decrease because a 4-way set associative 16 KW cache with a 4 KW page size can use physical tags which require fewer bits than virtual tags (20 versus 32).

The performance gain from modifying the base architecture in the way we have described above is shown by the difference between the first and second columns of Figure 5.5. The performance of the improved architecture is 25 % better than the base architecture. Most of the performance increase is a product of the reduced cycle time (3.5 ns versus 5.2 ns) that the improved organization permits. However, the time spent in the memory system has been reduced from 2.1 ns per instruction to 1.6 ns per instruction which is a 25 % increase in memory system performance. Figure 5.6 shows a comparison between the CPI of the base architecture and the CPI of the improved architecture. This figure shows that the CPI of the improved architecture is 12 % worse than the base architecture. This increase is due to the fact that the speed of the L2 cache and main memory do not scale with a decrease in cycle time

[1] tag memory = L1-I tags + L1-D tags = $\frac{16\,\mathrm{KW}\times 32\,\mathrm{b}}{32\,\mathrm{W}} + \frac{4\,\mathrm{KW}\times 20\,\mathrm{b}}{8\,\mathrm{W}} = 26\,\mathrm{Kb}$

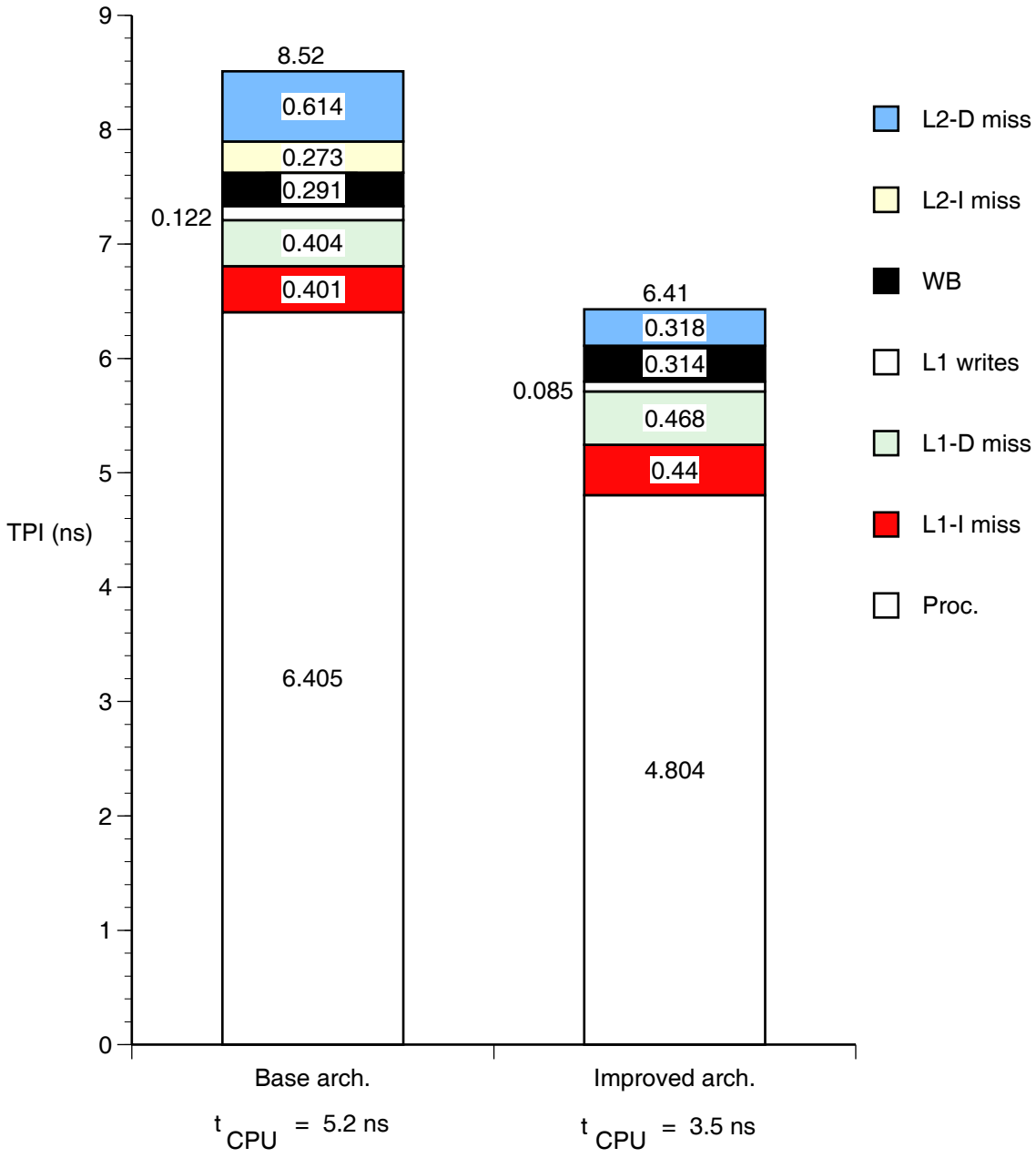

Figure 5.5: The performance gain from the improved architecture.

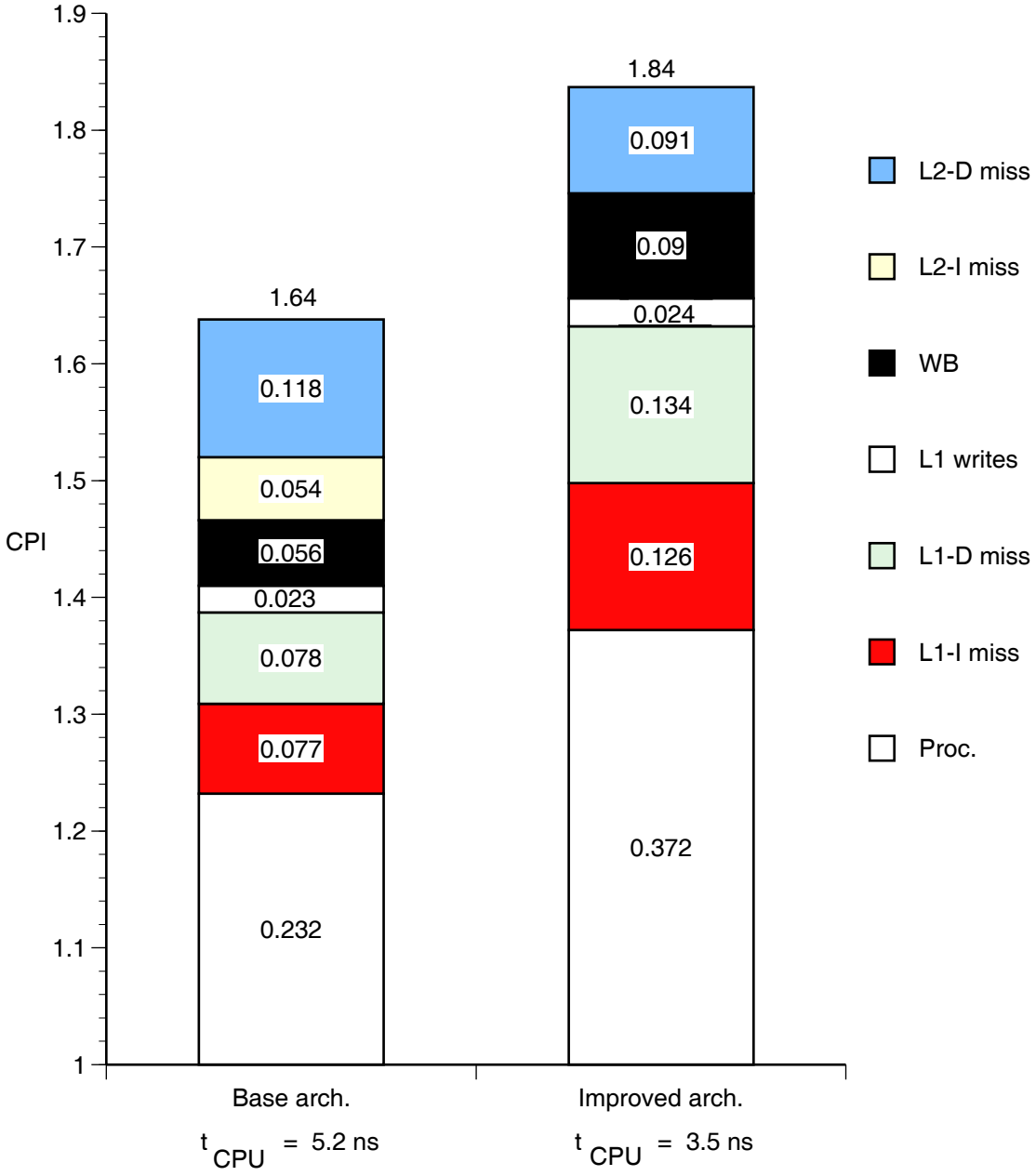

Figure 5.6: A comparison between the CPI of the base architecture and the improved architecture. Note that in the improved architecture there are no L2-I misses because the I-cache is single level.

and that more cycles are wasted from the extra branch and load delay slots in the improved architecture.

5.2 Write Policy

In this section the effect of write policy on system performance is studied. Four write-policies are considered, they are: the *write-back* policy used in the L1 cache experiments, the *write-miss-invalidate* policy used in the base architecture, a new policy called *write-only*, and finally, *subblock placement* [HP90]. The write-back policy uses the four entry deep write-buffer defined in the base architecture. The other three write-through policies use a one word wide, eight entry deep write buffer.

In the write-back policy used in this study, writes that hit in the cache take two cycles. Writes that miss in the cache take one cycle to check the tags plus the refill penalty to fetch the new block into the cache (*write-allocate*) followed by a cycle to actually perform the write into the cache. If the cache block that is replaced is dirty, the block is placed into the 4-entry-deep, block size wide write buffer. Finally, the tag bit that indicates that the new block is dirty is set. In the write-miss-invalidate policy, write hits take one cycle and write misses take two cycles. It is possible to complete write hits in a single cycle because the tag is checked while the data is written to the cache. If a miss is detected, a second cycle is used to invalidate the corrupted block. No previously written information is lost by doing this because all writes are sent to the write buffer. The write-only policy modifies write-miss-invalidate by updating the tag on a write-miss and marking the block as write-only. This allows one-cycle completion of subsequent writes to the block. All reads that map to a write-only block miss and cause the block to be reallocated.

In subblock placement each tag has four extra valid bits: one for each of the four words in the block. A write-miss causes the address portion of the tag to be updated in the next cycle. If the write was a word-write, the corresponding valid bit is turned on and all other bits are turned off. Subsequent word writes to the block update the valid bits in one cycle. However, partial word writes to the block do not

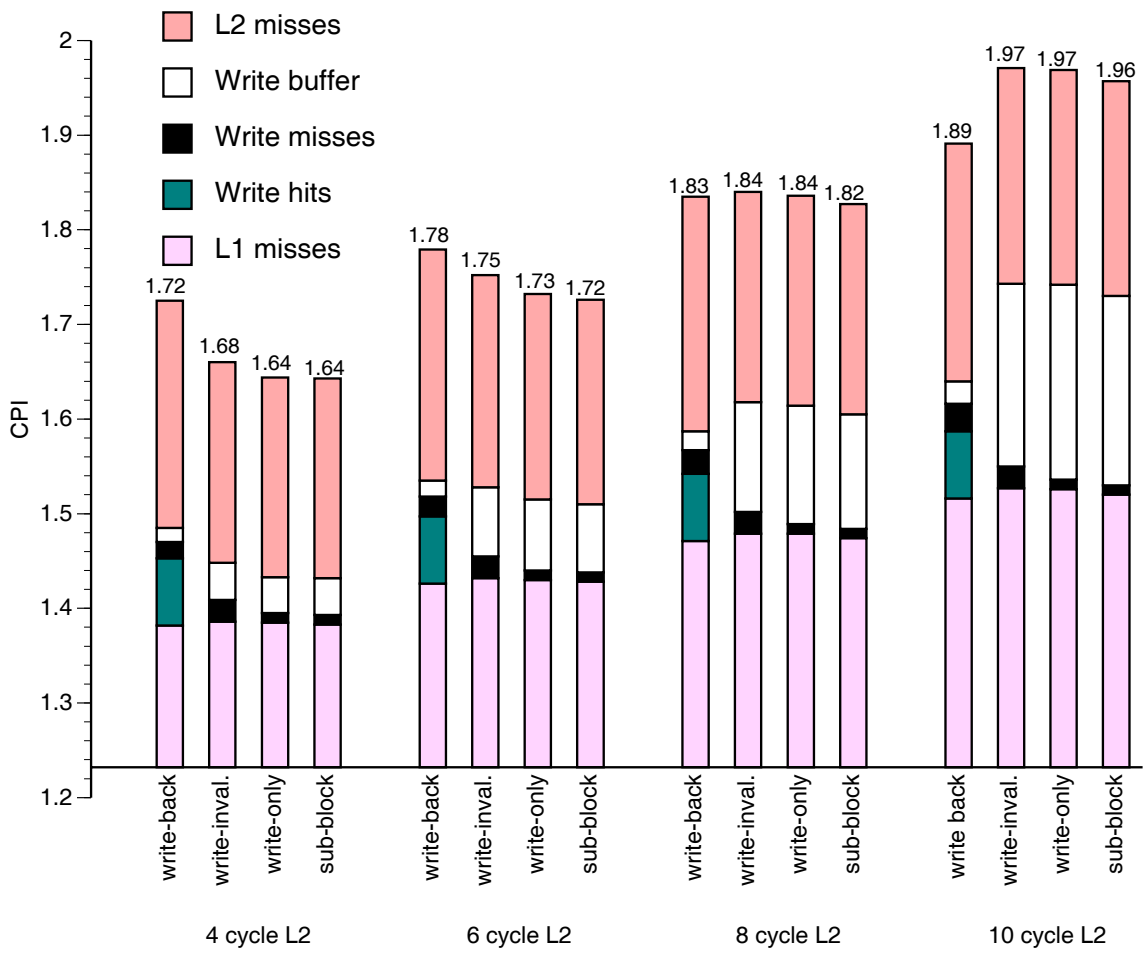

Figure 5.7: Write policy-L2 access time trade-off for the base architecture.

update the valid bits.

The performance of the base architecture using these four write-policies is compared based on CPI in Figure 5.7 for L2 access times of two to ten CPU cycles. These access times assume a two cycle latency to account for L2-tag checking and the communication delay between the L1 and L2 caches. Single reads and writes of L2 take the full access time. However, a stream of writes may overlap one or both cycles of latency. Figure 5.7 shows that for L2 access times of less than 8 cycles, write-through policies achieve higher performance, while for L2 access times greater than 8 cycles, the write-back policy achieves higher performance. The reason is that for a 4 KW cache, writes hit in the cache most of the time. Since writes make up a 0.0725 fraction of instructions, the constant 0.071 loss in CPI from 2-cycle write-hits

(*Write hits* in Figure 5.7) shown by the write-back policy indicates a hit rate of 98 %. In contrast, the write-through policies lose significantly less performance due to two-cycle-write-misses because the write-miss rate is only 2 %. However, write- through polices waste many more cycles on average waiting for the write buffer to empty before fetching the data for a L1 read miss than a write-back policy. The number of cycles it takes to empty the write-buffer is determined by the effective access time of the secondary cache. This assumes that changes in L2 cache size can be related to changes in effective L2 cache access time [Prz90a]. At some value of effective L2 access time the extra time spent waiting for the write-buffer in the write-through policies will be greater than the cost of two-cycle-write-hits in the write-back policy. This results in a trade-off between write policy and effective L2 cache access time. We note that the exact nature of this trade-off depends on the L1 read miss ratio and therefore on L1 size. However, the L2 access time at which a write-back policy becomes the better choice grows with L1 cache size because larger L1 caches have fewer read and write misses. This analysis of write-policies assumed a uniprocessor with a dedicated connection between the L1 and the L2 cache. If the L2 cache is shared, as it might be in a multiprocessor, contention for the L2 cache would increase its effective access time thereby favoring a write-back cache organization.

Figure 5.7 shows that for the region of L2 access time in which a write-through policy is most effective (4 and 6 cycles) write-only performs almost as well as subblock placement. The reason for this is that most of the performance gain (greater than 80 %) from subblock placement over write-miss-invalidate comes from writes misses that cause subsequent writes to hit. The extra performance gain from read hits that would otherwise have missed is less than 20 %. Write-only requires less tag memory than subblock placement (3 Kb for a 4 KW cache with a 4 W block size). Furthermore, write-only does not need the ability to read and write the tag RAM in the same cycle.

Using a write-through cache increases the number of slots in the write-buffer by a factor of two (8 deep instead of 4 deep), but decreases its width from four words to one word. The accompanying factor of four reduction in I/O requirements,

from 256 pins to 64 pins, may enables the write-buffer to be implemented on the CMU chip. Thus, a write-only policy provides higher performance and a cheaper and simpler implementation than a write-back policy for the L2 access times of the base architecture.

5.3 Memory System Concurrency

In this section we will evaluate techniques that improve performance by allowing more concurrency between memory accesses. The amount of concurrency will be limited to a modest increase in hardware complexity. Non-blocking data caches that allow more than a single miss to be outstanding at a time will also be investigated [Kro81, SF91].

With a split two-level data cache and a single-level instruction cache, instruction and data accesses are completely independent. Thus after an L1-I miss it is possible to refill the L1-I cache from the L2-I cache while the write-buffer continues to empty into the L2-D cache. This provides a decrease in CPI of 0.011.

To allow data-reads to bypass data-writes in the write-buffer usually requires that all eight entries of the write-buffer associatively match the address of the missed block. If a match is made, then all entries ahead, including the matched entry, must be flushed to keep the state of the L2-D cache consistent. However, if an extra dirty bit is added to the L1-D tags. The cache need only be flushed when dirty blocks are replaced in the cache. No associative matching is needed. This scheme works because the write-only policy ensures that all writes allocate a block in the L1-D cache. The write-buffer can only contain parts of dirty blocks. Therefore, to keep the L2-D cache consistent it is only necessary to flush the write-buffer when these blocks are replaced. Experiments show that this scheme achieves 95 % of the performance increase of associative matching. However, this performance increase is very modest; only a 0.012 decrease in CPI.

A large component of L2 cache performance loss is due to L2-D dirty misses. The reason for this is that when a L2-D block that is dirty must be replaced, the dirty

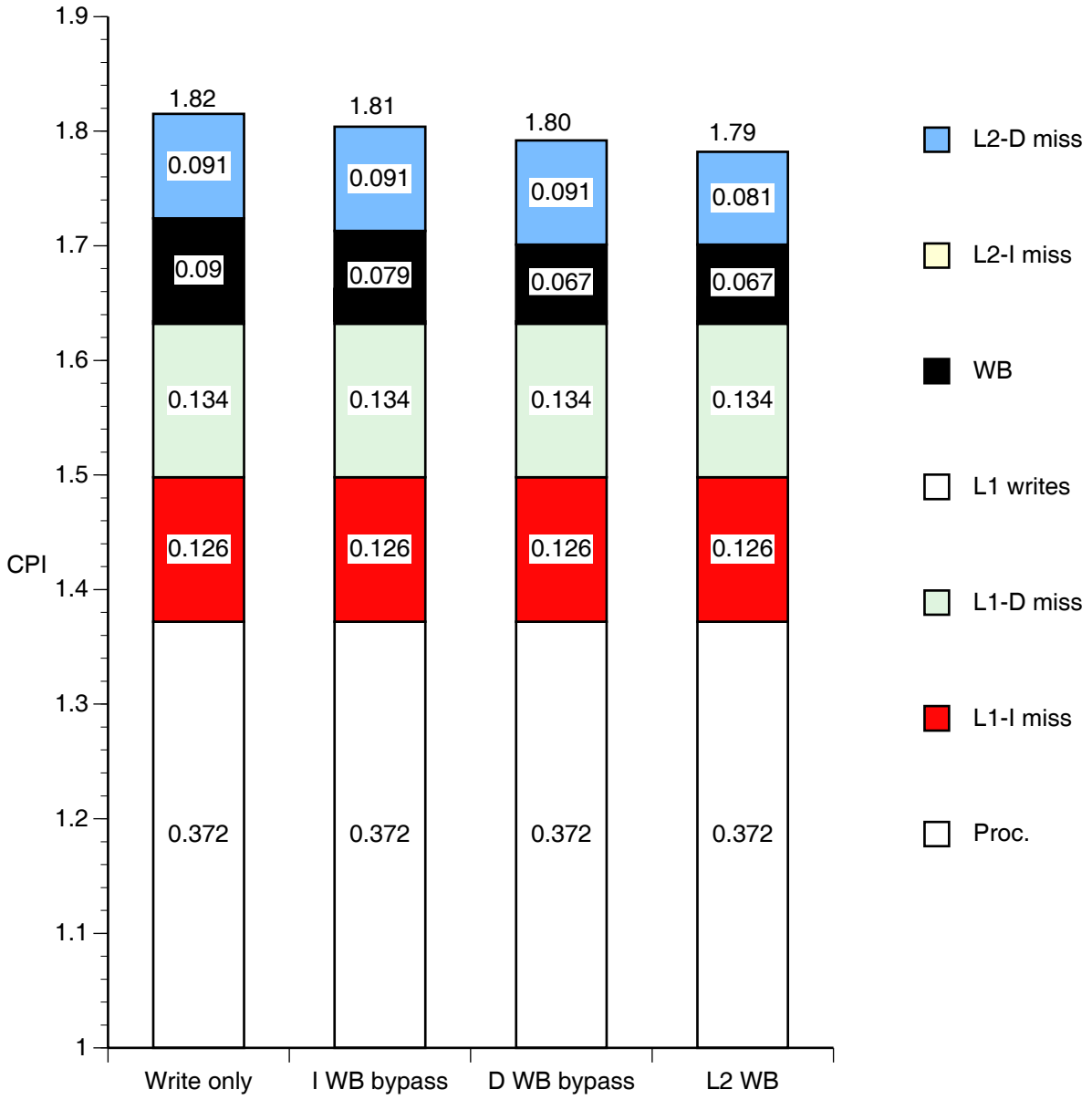

Figure 5.8: The performance improvement gained from adding more concurrency to the memory system.

block is first written to main memory and then the requested block is read from main memory which almost doubles the miss penalty over that of replacing a clean block. To reduce the L2 miss penalty, a single 32 word block write-buffer or dirty buffer can be added to the L2-D cache. Now it is possible to read the requested block before the dirty block is written. However, the performance improvement from doing this is only a 0.01 decrease in CPI. The reason that adding a dirty buffer does not decrease the CPI more is that most L2 cache misses are clustered at context switches. Therefore, the next miss usually has to wait for the entry in the buffer to written to main memory.

The total performance improvement from adding concurrency to the system is a 0.035 decrease in CPI. It is clear that increasing concurrency in the memory system in these limited ways adds much less to the performance of the memory system than the basic cache size, organization and speed optimizations discussed in the previous sections. In fact, it is questionable, whether the last optimization is worth implementing at all given the increase in control logic and memory that it requires. If an L2 dirty buffer is not implemented the optimized architecture that results from the improvements we have outlined in this chapter are diagrammed in Figure 5.9

Recently, non-blocking data caches have been proposed as a method for increasing the memory bandwidth of an L1-D cache. A non-blocking cache allows more than one cache miss to be serviced at the same time. This is done by allowing the processor to continue executing instructions after a load instruction that causes a cache miss. The processor will stall only if an instruction following load is dependent on the result of the load instruction or if the number of outstanding cache misses exceeds the capacity of the implementation of the non-blocking cache.

Figure 5.10 shows the potential benefit of a non-blocking cache. It shows that when there is more than one miss outstanding a non-blocking cache can improve performance. In this example the number of cycles it takes to execute the instruction sequence is reduced by two thirds by using a non-blocking cache. To be effective a non-blocking L1 cache requires a pipelined L2 cache so that new requests for data

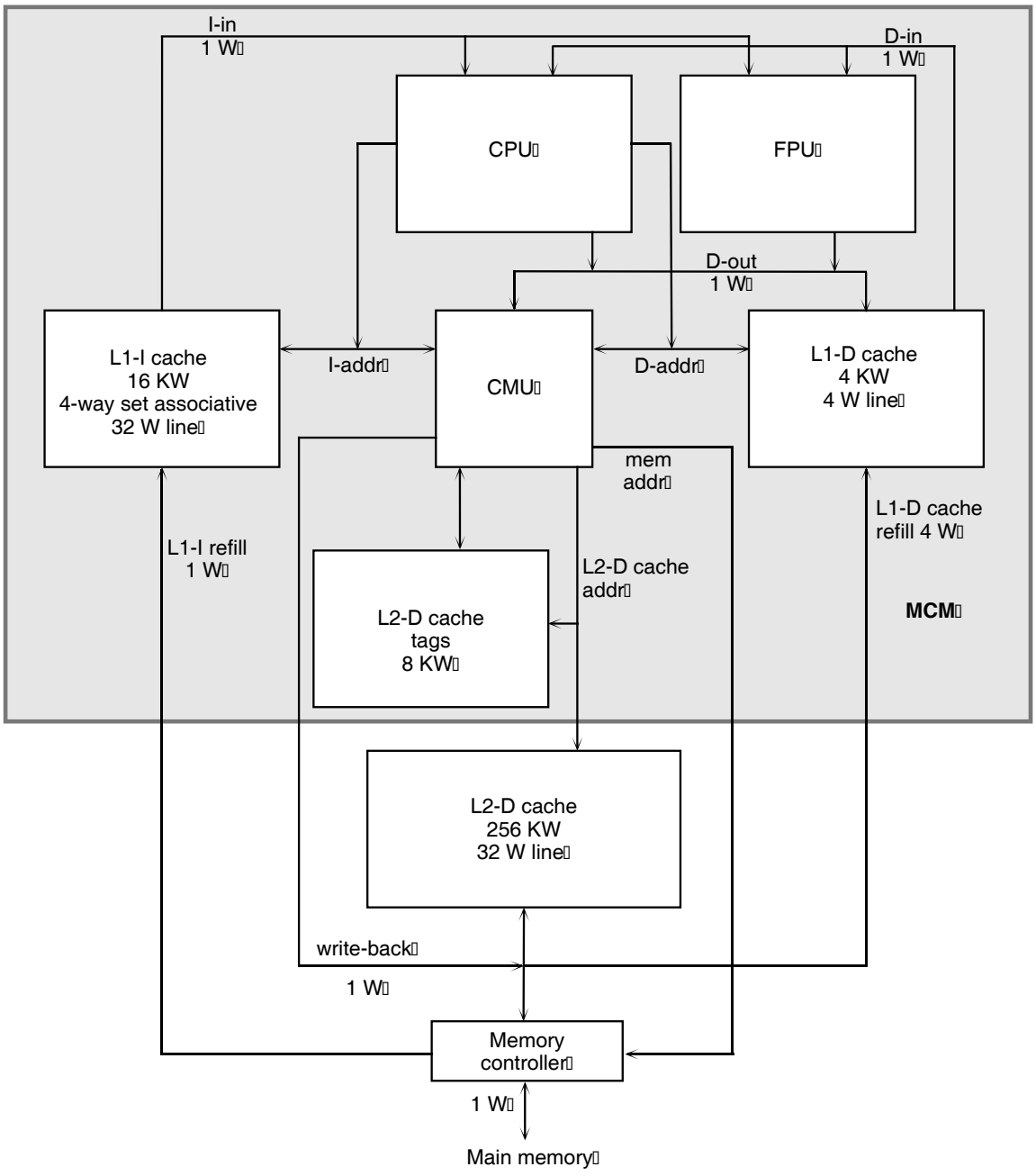

Figure 5.9: Optimized architecture.

instruction	Non-blocking cache	Blocking cache
L1	0	0
L2	1	10
L3	2	20
U1	10	30
U2	11	31
U3	12	32

Figure 5.10: Performance Improvement using a non-blocking cache. The instructions `L1`, `L2` and `L3` represent load instructions that cause cache misses. The instructions `U1`, `U2` and `U3` are instructions that use the results of the loads.The numbers under the heads represent the times at which each of the instructions are issued assuming a miss penalty is 10 CPU cycles.

can be sent to the L2 cache every cycle.

The implementation of a non-blocking cache presented in [Kro81] uses miss information/status holding registers (MSHRs) to hold the status information for outstanding misses. One MSHR is required for each outstanding miss. An MSHR that is associated with a miss contains the L1 cache index of the miss and the CPU destination register of the load instruction. This information is used to prevent other misses to the same L1 cache block from requesting data from the L2 cache and to send data from the L1 cache to the correct CPU register. In this implementation, a non-blocking cache requires the register file to be written after the normal execution of a load instruction has completed. To provide this capability, an extra port must be added to the register file. Scoreboarding logic is used to ensure that the processor will stall if an instruction requires the data from an outstanding miss.

We have developed a new method of estimating the performance improvement of non-blocking caches. This method can simulate a non-blocking cache configuration with any number of MSHRs and any miss penalty using a single pass of the trace. The method assumes that the L2 cache is pipelined and that it can accept a new request for data every cycle. The method can use any of the write-policies discussed in Section 5.2. However, to simplify the simulation, a write-back policy with

no-allocate on a write miss is assumed, but the effect of write buffering is ignored because it is small. The method assumes that the processor stalls in the event of an L2 miss.

L1 cache misses can be classified into two classes: 1) those that can be overlapped with previous L1 misses and 2) those that cannot be overlapped with previous L1 misses. L1 cache misses can be overlapped whenever there is a free MSHR and there is an outstanding miss. A miss is said to be outstanding between the time the load instruction that causes it is issued and the time an instruction that uses the result of the load is issued. For example, in Figure 5.10 a miss caused by `L1` is outstanding between `L1` and `U1`. Other misses, those of `L2` and `L3` can be overlapped with the `L1` miss during this period. However, if there are only two MSHRs, miss `L3` can not be overlapped and instead must be serviced after the data for miss `L1` has been returned by the L2 cache. To minimize the complexity and maximize the speed of a non-blocking cache implementation one should use the fewest number of MSHRs that provide adequate performance. Because these MSHRs must be searched associatively after each L1 cache miss. The classification procedure described below can determine the maximum number of MSHRs registers needed to support the maximum number of concurrent misses for any set of benchmarks and cache sizes.

Misses are classified using a FIFO queue. The entry at the head of the queue, h, contains the time at which the data from the most recently used miss was used. The tail of the queue, t, contains the time of the most recent miss. The number of entries o between the h and t is the number of outstanding misses. Given m, the number of MSHRs in a particular non-blocking cache implementation, if a miss occurs and $o < m$ the miss is classified as overlapped; otherwise, when $o \geq m$, the miss is classified as a non- overlapped miss. The time in clock cycles between a miss and t is called the degree of non-overlap n. The time in clock cycles between a load instruction that misses and an instruction that uses that result of the load is u.

For an overlapped miss the number of cycles given by the $\max(0, n - u)$ is the number of cycles that can not be hidden. For a non-overlapped miss the number of cycles given by u is the number of cycles that can be hidden if the CPU does not

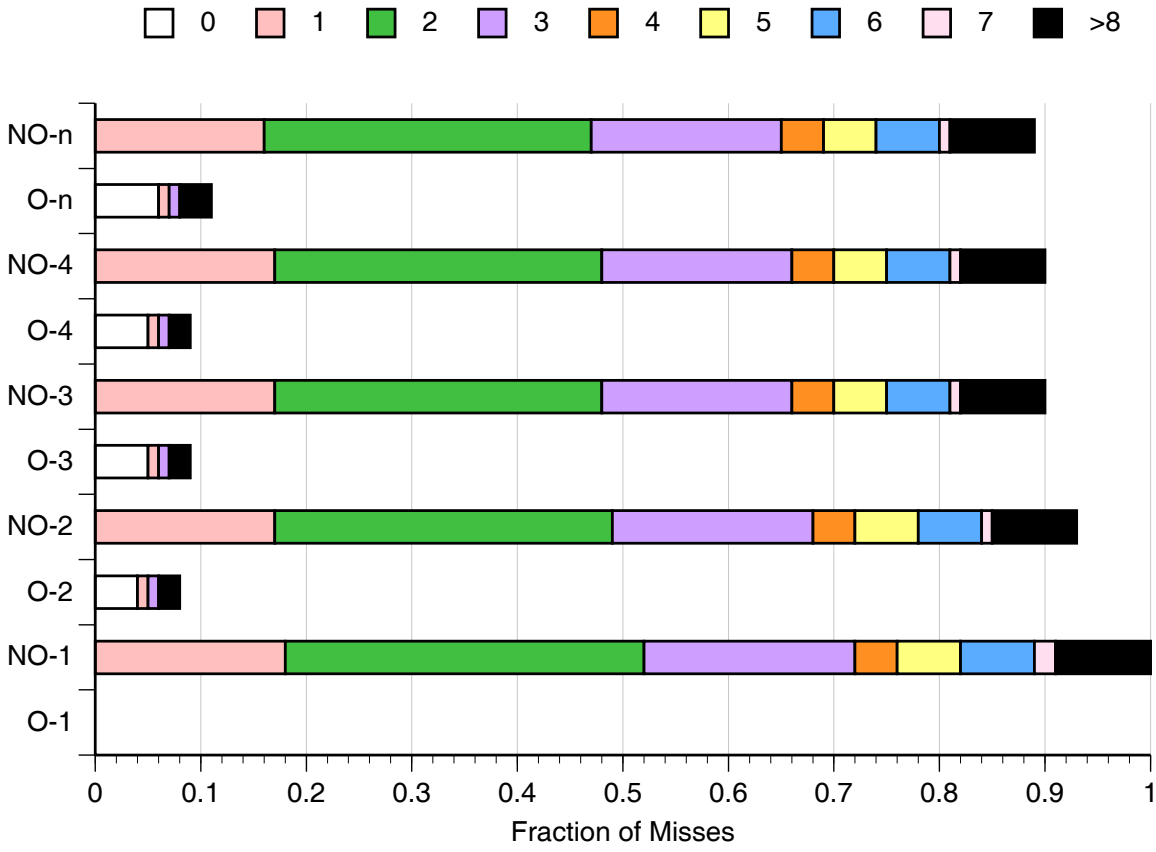

Figure 5.11: The performance of blocking and non-blocking 2 KW L1-D caches. Fraction of misses that are non-overlapped (NO) and overlapped (O) are shown for each number of MSHRs. The number of MSHRs appears after the dash in each label. The numbers by the grey levels represent the number of cycles that can be hidden for NO misses and the number of misses that cannot be hidden for O misses.

stall until an instruction that uses the result of a load, *i.e.*, the processor continues to fetch and execute instructions until the instruction that needs the cache-miss data is fetched. Doing this requires register scoreboarding logic, but does not require a non-blocking cache. Figures 5.11-5.13 plot the results of the simulation for blocking and non-blocking direct-mapped L1-D caches sizes that vary from 2 KW to 8 KW and for varying numbers of MSHRs. The number of MSHRs varies from 1, which corresponds to a blocking cache, to 4. To assess the limits in the performance increase from a non-blocking cache a configuration with an infinite number of MSHRs is also simulated. The results show that, at best, for architectures that execute one instruction per cycle, non-blocking caches can fully overlap only 10 % of the misses. Thus, with a

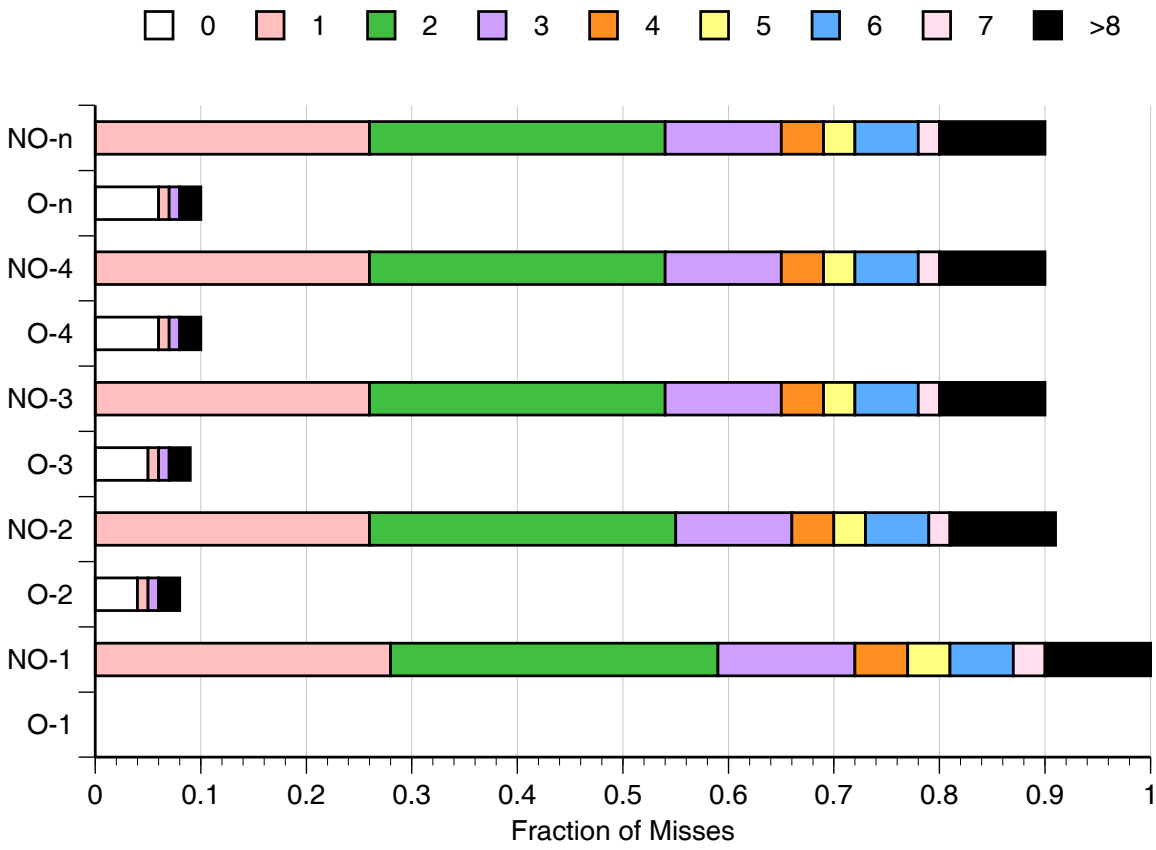

Figure 5.12: The performance of blocking and non-blocking 4 KW L1-D caches. Fraction of misses that are non-overlapped (NO) and overlapped (O) are shown for each number of MSHRs. The number of MSHRs appears after the dash in each label. The numbers by the grey levels represent the number of cycles that can be hidden for NO misses and the number of misses that cannot be hidden for O misses.

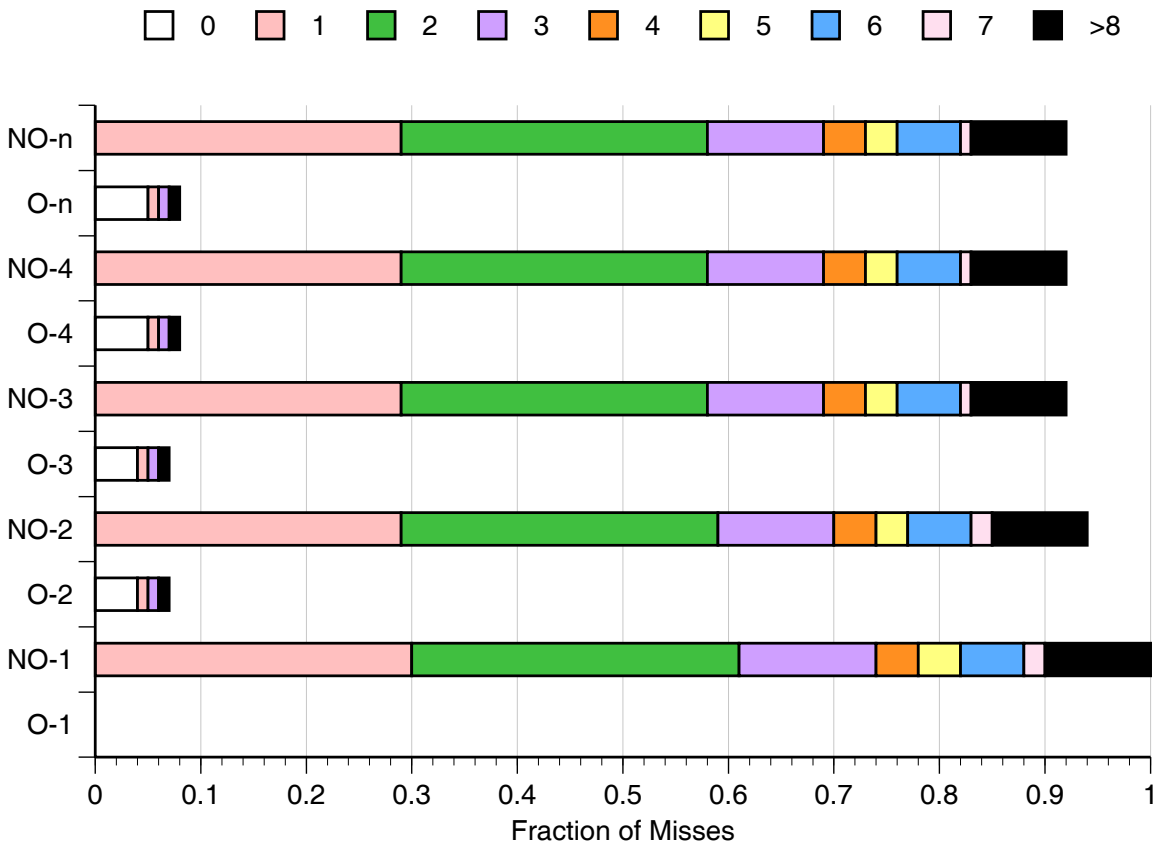

Figure 5.13: The performance of blocking and non-blocking 8 KW L1-D caches. Fraction of misses that are non-overlapped (NO) and overlapped (O) are shown for each number of MSHRs. The number of MSHRs appears after the dash in each label. The numbers by the grey levels represent the number of cycles that can be hidden for NO misses and the number of misses that cannot be hidden for O misses.

dynamic frequency of load instructions of 0.25, a L1-D cache miss ratio of 0.1, and a miss penalty of 10 CPU cycles, the decrease in CPI that would result from using a non-blocking cache is 0.025. This performance benefit is too small to justify the added complexity and possible increase in cycle time of a non-blocking cache. The figures also show that a non-blocking cache with 4 MSHRs is almost as good as a cache with an infinite number of MSHRs. The maximum number of MSHRs that were used in any of the benchmarks was 11.

5.4 Conclusions

This chapter has discussed two-level cache hierarchies for MCM based caches. Trace-driven simulation was used to explore the technology-organization trade-offs.

Experiments with four L2 cache organizations show that splitting the cache into instructions will reduce the performance of the L2 cache until the caches are large. However, an analysis of the speed-size trade-off for split caches reveals that for the L1-I cache the curves are almost flat for large caches. In contrast, for the L1-D cache, the curves are sloping downward for the largest cache size. This indicates the L2-I cache should be smaller and faster than the L2-D cache. Furthermore, small fast caches of the size that can feasibly be placed on the MCM provide higher performance than larger-slower L2-I caches that are placed off of the MCM. The opposite is true of an L2-D cache; here, larger, slower off-MCM caches provide higher performance.

The results of Chapter 4 show that larger and slower L1 caches are profitable when the L1 cache is pipelined. These results along with area and power constraints of the MCM technology are used to justify changing the base architecture to a single level 4-way set-associative 16 KW I-cache. The single level cache has the added advantage that the cost of L1-I misses is eliminated. The off-chip L2 cache is made into an L2-D cache. The number of branch delay slots is increased to three and the number of the load delay slots is increased to two. Making these changes to the base architecture leads to a substantial increase in performance.

Experiments that were conducted to evaluate the effect of changing L1 write-

policy and of increasing the concurrency in the cache- hierarchy. These experiments showed that, for CPU cycle times of 2 to 10 ns, where L1 miss penalties of under 10 cycles are expected, a write-only policy is better than a write-back policy. Relatively less performance is gained by using extra hardware to add concurrency to memory accesses through the use of a dirty buffer, through data-read conflict checking in the write buffer, or through the use of non-blocking L1-D caches.

CHAPTER 6

CONCLUSIONS

In this chapter we will summarize the results and the limitations of the design methodology, performance data and conclusions presented in this thesis. The limitations will suggest future directions for work.

6.1 Summary of Results

We have presented a multilevel-optimization design methodology for investigating the trade-offs that can be made between technology and organization of computer designs using time per instruction as the performance metric. This design methodology includes the use of delay-macro modeling, timing analysis and trace-driven simulation. New tools and analysis techniques were presented for all of these tasks. The design methodology has been extensively validated using a microprocessor design implemented using GaAs and MCM packaging technology. The design methodology begins with a base architecture and improves upon it by using simulation tools to exploring design alternatives.

Chapter 2 showed that when time per instruction is used as the performance metric, all architectural trade-offs can be expressed as trade-offs between the average number of cycles per instruction (CPI) and the cycle time (t_{CPU}).

Chapter 3 presented the multilevel design methodology and the associated performance analysis tools. Finally, Chapter 3 proposed a base architecture as a point of departure for the design trade-offs studied throughout the rest of the thesis. We have used timing analysis tools ($checkT_C$ and $minT_C$) to investigate the latency-t_{CPU}

tradeoff in the base architecture. This investigation showed that a substantial reduction in t_{CPU} can be achieved by increasing the pipeline depth of the BR-loop and the MEM-loop which contain the L1-D cache and L1-I cache respectively. To measure CPI we have used trace-driven simulation. Because the credibility of trace-driven simulation is dependent on the traces and the way in which the traces are simulated, we have used sixteen realistic benchmarks which are simulated in a multiprogramming environment. The one deficiency in the traces is the absence of operating system references. The values of CPI obtained from these simulations include all processor stalls from multicycle instructions such as loads, branches, and floating point operations, which are required for an accurate measure of performance.

Chapter 4 showed that the access time of on-chip and MCM caches can be estimated using simple expressions. These expressions were used to investigate the speed-size tradeoff for on-chip and MCM-based caches. The results showed that for a fixed cache size, smaller SRAM chips can provide lower overall access times, even though they take up more MCM area than larger SRAM chips. When the access times of MCM based caches are combined with timing analysis the results show that smaller SRAMS have the added advantage of balancing the MCM delay with the SRAM access time. This makes it possible to increase the pipeline depth of the cache loops and thus reduce t_{CPU}. The tradeoff between larger SRAMS with longer SRAM access times or smaller SRAMS with longer MCM delay favors smaller SRAM chips. Furthermore, smaller SRAMS have higher yield and thus lower cost which is very important in a an immature technology like GaAs DCFL.

Chapter 4 also shows that unless the datapath loops that contain the L1 caches are pipelined to a depth of two or more, the access time of on-chip or MCM based caches will cause the cycle time to be much longer than the minimum set by the ALU add time. However, if the pipeline depth of the cache loops is increased enough it is possible to ensure that the EX-loop will set the cycle time, even for large caches (32 KW). Increasing cache-loop pipeline depth increases the number of delay slots, which increases CPI.

The extra delay slots caused by increasing the pipeline depth of the cache

loops can be hidden by static compile time instruction scheduling schemes or dynamic hardware schemes. Experiments that were conducted to investigate both of these schemes for loads and branches show that static schemes work better at filling branch delay slots, but that dynamic schemes work better for filling load delay slots. The specific results for static branch prediction show that the increase in CPI due to the extra instruction cache misses caused by the larger code size should not be ignored, especially if the block size of the cache is small (4 W) or the refill rate is low (less than 1 W per cycle). Furthermore, with static schemes, the increase in CPI from up to three delay slots can effectively be hidden. When these results are combined with cache simulation and timing analysis, the results show that increasing the pipeline depth of the cache loops will increase the performance of a processor because it moves the optimal performance design point to lower cycle times and larger caches.

In Chapter 5, the organization of a two-level cache hierarchy was investigated. The results showed that split cache organizations are roughly equal to the unified cache performance for caches larger than 256 KW. Furthermore, the speed-size tradeoff for instruction and data caches is quite different. For highest performance, instruction caches should be smaller by a factor of eight and faster than data caches. Based on this information and the results of Chapter 4, a number of changes are made to the base architecture. The L1-I cache is increased in size, set-associativity and its access is more deeply pipelined. The L1-I cache is refilled directly from main memory. The L1-D cache is also pipelined and the L2 cache is made into a dedicated L2-D cache. Making these changes increases performance substantially.

Chapter 5 also presents the results of studies of L1-D cache write-policy, L2 cache speed-size trade-offs, and memory system concurrency. These results suggest that for L2 speeds of 8 cycles or less write-through policies provide higher performance, but for L2 speeds of greater than 8 cycles a write-back policy provides higher performance. The simulations that included modest increases in concurrency to the cache-hierarchy showed little performance improvement. Despite this small improvement, a new technique was proposed to allow loads to pass stores in the write buffer without the use of associative matching logic. This technique which only flushes the

write buffer when dirty lines in the L1-D cache are replaced achieves over 90 % of the performance of associative matching logic.

6.2 Limitations and Future Directions

In this thesis the focus has been on the cache hierarchy. Little has been said about the CPU or the FPU. We have argued that the pipeline depth of the EX-loop should not be increased because this would require latency values of five or more in the BR-loop and MEM-loop. At present, in GaAs, it is possible to perform a 32 b integer addition in under 2 ns and so pipelining the ALU would only be required to obtain GHz operation. However, in CMOS, which is a slower more dense technology increasing the pipeline depth of the EX-loop might improve performance. Furthermore, CMOS provides the integration density to implement the dynamic delay-slot hiding mechanisms that will surely be necessary to hide five or more delay slots.

Due to the higher integration densities available in CMOS technology, there are organizations that are realizable in CMOS that are not practical in GaAs. The most important of these is the ability to issue more than one scalar instruction in a cycle, which has been termed superscalar [Joh91b]. However, all studies that have been done of these organizations have largely neglected the implementation details and the effect that the multiple instruction issue logic, will have on cycle time. Nonblocking caches have been suggested to satisfy the data bandwidth requirements for superscalar processors. It is clear that due to the extra silicon real-estate that will be necessary to implement the multiple instruction issue and execution logic the sizes of the on-chip caches will be reduced compared to a scalar processor. Therefore, it would be interesting to extend the results of Chapters 4 and 5 by comparing the performance of a large pipelined off-chip cache with a small on-chip nonblocking cache.

There are two issues in trace driven cache simulation that need more attention. These are more realistic traces and the effect of context switching on cache performance. To accurately simulate the performance of real computer systems long traces that include operating system kernel references are a necessity. However, the

only way to get distortion free kernel references are with a hardware monitor. Unfortunately, most hardware monitors only collect short traces. Therefore getting long traces that include kernel references is a difficult problem. Chapter 3 showed that decreasing context switch times can reduce the performance of a cache significantly. Therefore, it would be interesting to characterize exactly the context switch rate of high performance processor in a multiprogramming environment, and effect it has on cache performance.

In conclusion, this thesis has demonstrated that exploring trade-offs between implementation technology and system organization can lead to higher performance computer designs and that the implementation technology of an architecture must be considered in order to provide accurate and credible performance predictions. It is my hope that the results contained in this thesis will lead other researchers to consider the trade-offs between implementation technology and organization.

APPENDIX A

A SUSPENS MODEL FOR GAAS

This appendix describes a SUSPENS model for GaAs which has been adapted from a SUSPENS model for NMOS that is described in [Bak90]. In the description of each of the calculation steps that follows the changes that were made to the NMOS model to accommodate GaAs are noted.

1. Based on the number of logic gates on a chip N_g, the average interconnection length $\overline{R}$ in units of gate pitch is calculated using Rent's rule.

$$\overline{R} = \frac{2}{9}\left(7\frac{N_g^{p-0.5}-1}{4^{p-0.5}-1} - \frac{1-N_g^{p-1.5}}{1-4^{p-1}}\right)\frac{1-4^{p-1}}{1-N_g^{p-1}}$$

   The Implementation dependent parameters which include the Rent's constants that are used in the equation above and throughout the model are listed in the table below.

Logic depth f_{ld}	Wire length Rent's constant p	Pin count Rent's constant β	Pin count Proportionality constant K
8–12	0.6	0.63	1.4

2. The interconnection pitch p_w and the number of wiring layers n_w an the average fan-out of the gates f_g combined with the average interconnection length are used to calculate the average gate dimension using:

$$d_g = \frac{f_g \overline{R} p_w}{e_w n_w}$$

where e_w is the efficiency of interconnections which is related to k the width-to-length (W/L) ratio of the input transistors by $e_w = 0.35 - 0.125k$.

3. The chip size D_c, and average interconnection length l_{av} are calculated using:

$$D_c = \sqrt{N_g d_g}$$

$$l_{av} = \overline{R} d_g$$

4. The input gate capacitance C_{gin} is calculated using

$$C_{gin} = kC_{tr}$$

where C_{tr} is the input capacitance of a minimum size input transistor.

5. So far we have stayed fairly close to the NMOS SUSPENS model from here on the model is changed to follow the delay and power model developed for Vitesse 1.0 micron Gallium Arsenide process in [SJ90].

6. The fan-out FO is determined from average number of gates that are driven f_g, the average interconnection length l_{av}, the transistor to interconnect load proportionality constant α_{int} and the W/L ratio of th gate k using

$$\mathrm{FO} = f_g + \frac{l_{av}\alpha_{int}}{k}$$

7. The average gate delay T_g from

$$T_g = k_0 + \mathrm{FO}k_1 + R_{int}C_{int}\frac{l_{av}^2}{2} + R_{int}l_{av}C_{gin}$$

where k_0 and k_1 are technology dependent delay parameters, R_{int} and C_{int} are the resistance and capacitance per unit length of interconnect. The first two terms of the equation above are the delay of driving the gates in the next stage and the interconnect between this stage and the next stage. The third term is the distributed-RC delay of the interconnect. The fourth term is the delay due to the resistance of the interconnect and the input capacitance of the gates of the next stage.

8. The clock frequency f_c is calculated using

$$f_c = \left(f_{ld} T_g + R_{int} C_{int} \frac{D_c^2}{2} + \frac{D_c}{v_c} \right)^{-1}$$

where v_c is the on-chip electromagnetic wave propagation speed. The first term of this equation is the delay of logic gates, the second term is the distributed-RC delay of a signal that travels halfway across the chip and the third term is the time-of-flight delay.

9. the number of pins N_p of the chip is calculated using Rent's rule

$$N_p = K N_g^{\beta}$$

10. The power consumption of the chip P_c is calculated using

$$P_c = N_g p_g k + \frac{1}{6} N_p I_{\mathrm{IO}} V_{EE}$$

where p_g is the power consumption of a gate with $k = 1$, I_{IO} is the output pad current and V_{EE} is the power supply voltage. The first term of this equation is the power consumption of the gates, the second term is the power consumption of the I/O pads.

11. The technology dependent parameters used in the GaAs SUSPENS model are listed below:

Parameter	Value
p_w	3.5 μm
n_w	4
f_g	3
C_{tr}	1.62 fF
k_0	18.2 ps
k_1	46.7 ps
R_{int}	0.046 Ω/μm
C_{int}	0.27 fF/μm
p_g	8.9 μA/μm
I_{IO}	20 mA
V_{EE}	2 V

BIBLIOGRAPHY

[AF88] D. B. Alpert and M. J. Flynn, "Performance trade-offs for microprocessor cache memories," *IEEE MICRO*, vol. 8, Aug. 1988.

[Aga88] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Publishers, 1988.

[Agr82] V. D. Agrawal, "Synchronous path analysis in MOS circuit simulator," in *Proc. 19th ACM/IEEE Design Automation Conf.*, pp. 629–635, 1982.

[AHH88] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Trans. Computer Systems*, vol. 6, pp. 393–431, Nov. 1988.

[AHH89] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM Trans. Computer Systems*, vol. 7, May 1989.

[AKCB86] C. Alexander, W. Keshlear, F. Cooper, and F. Briggs, "Cache memory performance in a UNIX environment," *Computer Architecture News*, vol. 14, pp. 41–70, June 1986.

[ALBL91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazosska, "The interaction of architectures and operating system design," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 108–121, Apr. 1991.

[ASH86] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A new technique for capturing address traces using microcode," in *Proc.13th Annual Int. Symp. Computer Architecture*, pp. 119–129, 1986.

[Bak87] H. B. Bakoglu, "A system level circuit model for multiple and single-chip CPUs," in *IEEE Int. Solid State Circuits Conf.*, pp. 308–309, Feb. 1987.

[Bak90] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

[BC91] D. Bhandarkar and D. W. Clark, "Performance from architecture: Comparing a RISC and CISC with similar hardware organization," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 310–319, Apr. 1991.

[BKB90] H. O. Bugge, E. H. Kristiansen, and B. O. Bakka, "Trace-driven simulations for a two-level cache design in open bus systems," in *Proc.17th Annual Int. Symp. Computer Architecture*, pp. 250–259, June 1990.

[BKW90] A. Borg, R. E. Kessler, and D. W. Wall, "Generation and analysis of very long address traces," in *Proc.17th Annual Int. Symp. Computer Architecture*, pp. 270–279, June 1990.

[BW87] J.-L. Baer and W.-H. Wang, "Architectural choices for multilevel cache hierarchies," in *Proc. Int. Conference on Parallel Processing*, pp. 258–261, unknown 1987.

[BW88] J.-L. Baer and W.-H. Wang, "On the inclusion properties for multi-level cache hierarchies," in *Proc.15th Annual Int. Symp. Computer Architecture*, pp. 73–80, June 1988.

[BWL89] J.-L. Baer, W.-H. Wang, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Proc.16th Annual Int. Symp. Computer Architecture*, pp. 140–148, June 1989.

[CBK88] D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring vax 8800 performance with a histogram hardware monitor," in *Proc.15th Annual Int. Symp. Computer Architecture*, pp. 176–185, June 1988.

[CCH+87] F. Chow, S. Correll, M. Himestei, E. Killian, and L. Weber, "How many addressing modes are enough?," in *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 117–121, Oct. 1987.

[Che88] J. J. Cherry, "Pearl: A CMOS timing analyzer," in *Proc. 25th ACM/IEEE Design Automation Conf.*, pp. 148–153, 1988.

[Cla83] D. W. Clark, "Cache performance in the VAX-11/780," *ACM Trans. Computer Systems*, vol. 1, pp. 24–37, Feb. 1983.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Intoduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[Dag87] M. R. Dagenais, *Timing Analysis for MOSFETs: An Integrated Approach*. PhD thesis, McGill University, 3480 University Street, Montreal, Quebec, Canada H3A 2A7, June 1987.

[DF90] P. K. Dubey and M. J. Flynn, "Optimal pipelining," *Jour. Parallel and Distributed Computing*, vol. 8, Jan. 1990.

[DR89] M. R. Dagenais and N. C. Rumin, "On the calculation of optimal clocking parameters in synchronous circuits with level–sensitive latches," *IEEE Trans. Computers*, vol. 8, pp. 268–278, Mar. 1989.

[Dun86] R. R. Duncombe, "The SPUR instruction unit: An on-chip instruction cache memory for a high performance VLSI multiprocessor," Technical Report UCB/CSD 87/307, Computer Science Division, University Of California, Berkeley, Aug. 1986.

[Dyk90] J. A. Dykstra, *High-Speed Microprocessor Design With Gallium Arsenide Very Large Scale Integrated Digital Circuits.* PhD thesis, University of Michigan, Ann Arbor, 1990.

[ED87] P. G. Emma and E. S. Davidson, "Characterization of branch and data dependencies in programs for evaluating pipeline performance," *IEEE Trans. Computers*, vol. C-36, July 1987.

[EF78] M. C. Easton and R. Fagin, "Cold-start vs. warm-start miss ratios," *Communications of ACM*, vol. 21, pp. 866–872, Oct. 1978.

[Faw75] B. K. Fawcett, "Maximal clocking rates for pipelined digital systems," Master's thesis, Dept. Elec. Eng., University of Illinois, Urbana-Champaign, 1975.

[GD85] L. A. Glasser and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits.* Reading, Massachusetts: Addison-Wesley, 1985.

[HCC89] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc.16th Annual Int. Symp. Computer Architecture*, pp. 224–233, June 1989.

[HCHU91] T. Hoy, A. Chandra, T. R. Huff, and R. Uhlig, "The design of a GaAs microprocessor." 1991 VLSI Contest Report, May 1991.

[HCLC91] R. S. Hinds, S. R. Canaga, G. M. Lee, and A. K. Choudhury, "A 20K GaAs array with 10K of embedded SRAM," *IEEE Jour. of Solid-State Circuits*, vol. 26, pp. 245–256, Mar. 1991.

[HCS+87] M. Horowitz, P. Chow, D. Stark, R. T. Simoni, A. Salz, S. Przybylski, J. Hennesssy, G. Gulak, and A. A. J. M. Acken, "MIPS-X a 20-MIPS peak, 32-bit microprocessor with on-chip cache," *IEEE Jour. of Solid-State Circuits*, vol. SC-22, pp. 790–799, Oct. 1987.

[Hil87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, 1987.

[Hil88] M. D. Hill, "A case for direct mapped caches," *Computer*, vol. 21, pp. 25–40, Dec. 1988.

[Hit82] R. B. Hitchcock, Sr., "Timing verification and the timing analysis program," in *Proc. 19th ACM/IEEE Design Automation Conf.*, pp. 594–604, 1982.

[HP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[HS84] M. D. Hill and A. J. Smith, "Experimental evaluation of on-chip microprocessor cache memories," in *Proc.11th Annual Int. Symp. Computer Architecture*, pp. 158–166, June 1984.

[HS89] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," Computer Sciences Technical Report 823, Computer Sciences Department, University of Wisconsin, Madison, Feb. 1989.

[Joh91a] M. Johnson, "Private communication." May 1991.

[Joh91b] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice Hall, Inc., 1991.

[Jou84] N. P. Jouppi, *Timing Verification and Performance Improvement of MOS VLSI Designs*. PhD thesis, Stanford University, Stanford, CA 94305–2192, Oct. 1984.

[Jou87] N. P. Jouppi, "Timing analysis and performance improvement of MOS VLSI designs," *IEEE Trans. Computer-Aided Design*, vol. CAD–6, pp. 650–665, July 1987.

[Jou89] N. P. Jouppi, "A 20-MIPS sustained 32-bit CMOS microprocessor with high ratio of sustained to peak performance," *IEEE Jour. of Solid-State Circuits*, vol. 24, pp. 1348–1359, Oct. 1989.

[Jou90] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc.17th Annual Int. Symp. Computer Architecture*, pp. 364–373, June 1990.

[Kan87] G. Kane, *MIPS R200 RISC Architecture*. Englewood Cliffs, New Jersey: Prentice Hall, 1987.

[Kat85] M. G. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*. Cambridge, Massachusetts: THE MIT Press, 1985.

[KD78] B. Kumar and E. S. Davidson, "Performance evaluation of highly concurrent computers by deterministic simulation," *Communications of ACM*, vol. 21, pp. 904–913, Nov. 1978.

[KH87] D. Kiefer and J. Heightley, "CRAY-3 a GaAs implemented supercomputer system," in *Proc. of GaAs Integrated Circuits Symposium*, 1987.

[Kro81] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 8th Annual Int. Symp. Computer Architecture*, pp. 81–87, June 1981.

[KS86] S. R. Kunkle and J. E. Smith, "Optimal pipelining in supercomputing," in *Proc.13th Annual Int. Symp. Computer Architecture*, pp. 404–411, June 1986.

[KSH+91] A. I. Kayassi, K. A. Sakallah, T. Huff, R. B. Brown, T. N. Mudge, and R. J. Lomax, "The effect of mcm technology on system performance," in *1991 Multichip Module Workshop— Extended Abstract Volume*, Mar. 1991.

[KT91] M. Katevenis and N. Tzartzanis, "Reducing the branch penalty by rearranging instructions in a double-width memory," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 15–27, Apr. 1991.

[LB90] S. I. Long and S. E. Butner, *Gallium Arsenide Digital Integrated Circuit Design*. New York, New York: McGraw-Hill Publishing Co., 1990.

[Lil88] D. J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer Magazine*, vol. 21, pp. 47–55, July 1988.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocesoosr," in *Proc.17th Annual Int. Symp. Computer Architecture*, pp. 148–159, May 1990.

[LPI86] S. Laha, J. H. Patel, and T. K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *IEEE Trans. Computers*, vol. 37, pp. 1325–1336, Nov. 1986.

[LS84] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer Magazine*, vol. 17, pp. 6–22, Jan. 1984.

[LS88] S. I. Long and M. Sundarm, "Noise-margin limitations on Gallium Arsenide VLSI," *IEEE Jour. of Solid-State Circuits*, vol. 23, pp. 893–900, Aug. 1988.

[LT88] T. Lovett and S. S. Thakkar, "The Symmetry multiprocessor system," in *Proc. Int. Conference on Parallel Processing*, p. unknown, unknown 1988.

[MBB+91] T. N. Mudge, R. B. Brown, W. P. Birmingham, J. A. Dykstra, A. I. Kayssi, R. J. Lomax, O. A. Olukotun, K. A. Sakallah, and R. Millano, "The design of a micro-supercomputer," *Computer*, vol. 24, Jan. 1991.

[MC80] C. Meade and L. Conway, *Introduction to VLSI Design*. Reading, Massachusetts: Addison-wesley, 1980.

[McM72] F. H. McMahon, "FORTRAN CPU performance analysis," tech. rep., Lawrence Livermore Laboratories, 1972.

[Met90] Meta-software., *HSPICE User's Manual H9001*, 1990.

[MF88] C. L. Mitchell and M. J. Flynn, "A workbench for computer architects," *IEEE Design and Test of Computers*, pp. 19–29, Oct. 1988.

[MGS+70] R. L. Mattson, J. Gecsei, J. Slutz, D. R., and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[MH86] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proc.13th Annual Int. Symp. Computer Architecture*, pp. 396–403, june 1986.

[MIP88] MIPS Computer Systems, Inc, *MIPS RISCompiler Languages Programmer's Guide*, Dec. 1988.

[MK90] M. McFarland and T. J. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 938–950, Sept. 1990.

[Mur83] K. G. Murty, *Linear Programming.* John Wiley & Sons, Inc., 1983.

[Nag90] D. Nagle, "A floating point simulator." Internal Research Report, May 1990.

[Ong84] D. G. Ong, *Modern MOS Technology— Processes, Devices, & Design.* New York, New York: McGraw-Hill Book Company, 1984.

[Ous85] J. K. Ousterhout, "A switch-level timing verifier for digital MOS VLSI," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 336–349, July 1985.

[PHH89] S. Przybylski, M. Horowitz, and J. Hennessy, "The performance impact of block sizes and fetch strategies," in *Proc.16th Annual Int. Symp. Computer Architecture*, pp. 114–121, June 1989.

[Prz90a] S. A. Przybylski, *Cache and Memory Hierarchy Design.* San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[Prz90b] S. A. Przybylski, "The performance impact of block sizes and fetch strategies," in *Proc.17th Annual Int. Symp. Computer Architecture*, pp. 160–169, June 1990.

[PS77] B. L. Peuto and L. J. Shustek, "An instruction timing model of CPU performance," in *Proc. 4th Annual Int. Symp. Computer Architecture*, pp. 165–178, unknown 1977.

[PS82] D. Patterson and C. Séquin, "A VLSI RISC," *IEEE Computer Magazine*, vol. 15, pp. 8–21, Sept. 1982.

[Puz85] T. R. Puzak, *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts, 1985.

[Rad69] C. Radke, "A justification of and improvement on a useful rule for predicting circuit to pin ratios," in *Proc. 9th ACM/IEEE Design Automation Conf.*, pp. 257–267, 1969.

[Rec89] J. H. Reche, "High density multichip interconnect for advanced packaging," in *NEPCON West Proceedings*, pp. 1308–1318, Mar. 1989.

[RTL90] D. Roberts, G. Taylor, and T. Layman, "An ECL RISC microprocessor designed for two-level cache," in *Proc. IEEE COMPCON*, p. 489, Feb. 1990.

[Rus78] R. M. Russell, "The CRAY-1 computer system," *Communications of ACM*, vol. 21, pp. 63–72, Jan. 1978.

[SCH+91] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, "Instruction level profiling and evaluation of the IBM RS/6000," in *Proc.18th Annual Int. Symp. Computer Architecture*, pp. 180–189, May 1991.

[SF91] G. Sohi and M. Franklin, "High bandwidth data memory systems for superscalar processors," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 53–62, Apr. 1991.

[SH87] H. Scales and P. Harrod, "The design and implementation of the MC68030 cache memories," in *Proc. IEEE Int. Conf. Computer Design*, pp. 578–581, unknown 1987.

[SJ90] S. Stritter and M. Johnson, "Preliminary benchmark of Vitesse GaAs." MIPS Computer Systems Report, Feb. 1990.

[SL88] R. T. Short and H. M. Levy, "A simulation study of two-level caches," in *Proc.15th Annual Int. Symp. Computer Architecture*, pp. 81–88, June 1988.

[SLL89] D. P. Seraphim, R. C. Lasky, and C.-Y. Li, *Principles of Electronic Packaging*. New York, New York: McGraw-Hill, Inc., 1989.

[SM91] H. Sachs and H. McGhan, "Future directions in Clipper processors," in *Proc. IEEE COMPCON SPRING '91*, pp. 241–246, Mar. 1991.

[Smi77] A. J. Smith, "Two methods for the efficient analysis of memory address trace data," *IEEE Trans. Software Engineering*, vol. SE-3, pp. 94–101, Jan. 1977.

[Smi81] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annual Int. Symp. Computer Architecture*, pp. 135–147, July 1981.

[Smi82] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, Sept. 1982.

[Smi85a] A. J. Smith, "Cache evaluation and the impact of workload choice," in *Proc.12th Annual Int. Symp. Computer Architecture*, pp. 64–73, June 1985.

[Smi85b] A. J. Smith, "Problems, directions and issues in memory hierarchies," in *Proc. 18th Annual Hawaii Conf. System Sciences*, pp. 468–476, Dec. 1985.

[Smi87] A. J. Smith, "Line (block) size choice for CPU cache memories," *IEEE Trans. Computers*, vol. C-36, pp. 1063–1075, Sept. 1987.

[SMO90a] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "Analysis and design of latch–controlled synchronous digital circuits," in *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990.

[SMO90b] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "$checkT_c$ and $mint_c$: Timing verification and optimal clocking of synchronous digital circuits," in *Proc. IEEE Conf. Computer-Aided Design*, (Santa Clara, California), Nov. 1990.

[SMO90c] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "A timing model of synchronous digital circuits," Technical Report CSE–TR–47–90, University of Michigan, Dept of EECS, Ann Arbor, MI 48109–2122, 1990.

[Sto90] H. S. Stone, *High-Performance Computer Architecture*. Addison-Wesley Publishing Co., 1990.

[SWP86] J. E. Smith, S. Weiss, and N. Y. Pang, "A simulation study of decoupled architecture computers," *IEEE Trans. Computers*, vol. 35, pp. 692–702, Aug. 1986.

[Szy86] T. G. Szymanski, "LEADOUT : A static timing analyzer for MOS cicuits," in *Proc. IEEE Conf. Computer-Aided Design*, pp. 130–133, 1986.

[TDF90] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice— a low-cost high-speed address translation mechanism," in *Proc.17th Annual Int. Symp. Computer Architecture*, pp. 355–363, June 1990.

[Tho63] J. E. Thornton, *Considerations In Computer Design— Leading Up To The CONTROL DATA 6600*. Control Data Corp., 1963.

[Tho90] M. Thorson, "ECL bus controller hits 266 Mbytes/s," *Microprocessor Report*, vol. 4, pp. 12–13, Jan. 1990.

[TS71] I. L. Traiger and D. R. Slutz, "One-pass technique for the evaluation of memory hierarchies," Technical Report RJ 892, IBM Research, July 1971.

[TS89] J. G. Thompson and A. J. Smith, "Efficient (stack) algorithms for analysis of write-back and sector memories," *ACM Trans. Computer Systems*, vol. 7, Feb. 1989.

[Tum91] R. R. Tummala, "Electronic packaging in the 1990's— a perspective from America," *IEEE Trans. Components, Hybrids and Manufacturing Technology*, vol. 14, pp. 262–271, June 1991.

[UT86] S. H. Unger and C.-J. Tan, "Clocking schemes for high–speed digital systems," *IEEE Trans. Computers*, vol. C-35, pp. 880–895, Oct. 1986.

[VIT89] VITESSE Semiconductor Corp., *Gallium Arsenide Integrated CFoundry Design Manual Version 4.0*, Mar. 1989.

[VIT91] VITESSE Semiconductor Corp., *Foundry Design Manual Version 5.0*, Mar. 1991.

[Wan89] N. Wang, *Digital MOS Integrated Circuits: Design for Applications*. Englewood Cliffs, New Jersey: Prentice Hall, 1989.

[WBL89] W.-H. Wang, J.-L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 140–148, June 1989.

[WF82] S. Wasser and M. Flynn, *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehart and Winston, 1982.

[Wil87] A. W. Wilson, Jr., "Hierarchical cache/bus architectures for shared memory multiprocessors," in *Proc.14th Annual Int. Symp. Computer Architecture*, pp. 244–252, June 1987.

[WMF89] D. Wong, G. D. Micheli, and M. Flynn, "Inserting active delay elements to achieve wave pipelining," in *Proc. IEEE Conf. Computer-Aided Design*, pp. 270–273, 1989.

[WS88] D. E. Wallace and C. H. Séquin, "ATV: An abstract timing verifier," in *Proc. 25th ACM/IEEE Design Automation Conf.*, pp. 154–159, 1988.