REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1996	

4. TITLE AND SUBTITLE
ContinuousuOvernight Observation of Human Tooth Eruption

5. FUNDING NUMBERS

6. AUTHOR(S)
Ronald K. Risinger

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student Attending:
North Caroling University

8. PERFORMING ORGANIZATION REPORT NUMBER
96-010

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPARTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET, BLDG 125
WRIGHT-PATTERSON AFB OH 45433-7765

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release IAW AFR 190-1
Distribution Unlimited
BRIAN D. GAUTHIER, MSgt, USAF
Chief Administration

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

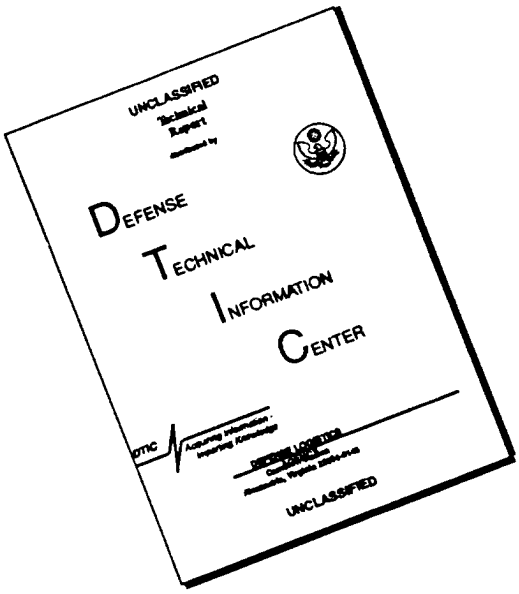
14. SUBJECT TERMS

15. NUMBER OF PAGES
83

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

★U.S.GPO:1990-0-273-271

A Memory Architecture to Support Real-Time Computer Systems

by

Michael Kevin Jack Milligan, BSEE, MSEE, MBA

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

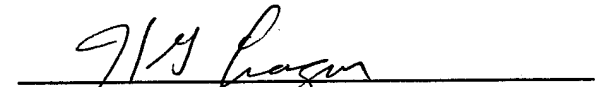
The University of Texas at Austin

May 1996

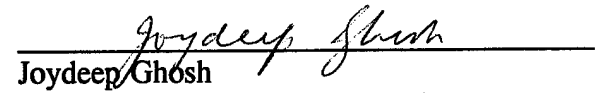
A Memory Architecture to Support

Real-Time Computer Systems

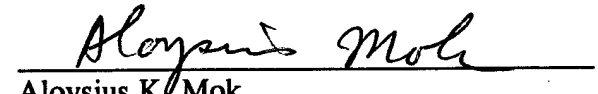
Approved by
Dissertation Committee:



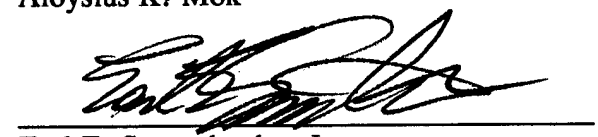
Harvey G. Cragon, Supervisor



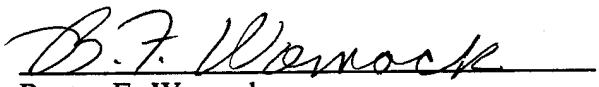
Joydeep Ghosh



Aloysius K. Mok



Earl E. Swartzlander, Jr.



Baxter F. Womack

Dedication

To my parents Jack and Clara Milligan, and to my family - Alison, Matthew and Kyle - for their support and encouragement.

Acknowledgments

I'd like to give special thanks to Professor Harvey G. Cragon for sharing with me a portion of his extensive knowledge of computer architecture. His dedication of time and effort in supporting my academic and professional goals are greatly appreciated.

I'd also like to thank committee members Joydeep Ghosh, Al Mok, Earl Swartzlander and Baxter Womack for their guidance. A special thanks to Professor Swartzlander for supporting my research through the use of his Sun SPARC laboratory.

I'd like to thank the United States Air Force and especially Colonel Alan Klayton, head of the Electrical Engineering Department at the U.S. Air Force Academy, for giving me the opportunity to pursue the Ph.D. degree.

A special thanks to my wife Alison for her support and patience.

A Memory Architecture to Support

Real-Time Computer Systems

Publication No. _____

Michael Kevin Jack Milligan, Ph.D.

The University of Texas at Austin, 1996

Supervisor: Harvey G. Cragon

Real-time computer systems must meet specific deadlines in generating their outputs. As a result, all components of the real-time system must have predictable performance so hardware resources can be effectively scheduled, ensuring all deadlines are met. Hierarchical memories, by their nature, have components with unpredictable behavior. For this reason, hierarchical memory subsystems are often not used in hard real-time systems and the substantial performance advantages they offer are therefore not realized. This research focuses on developing methods which make the highest level member of the memory hierarchy - the cache - predictable as seen by the processor, so it may be used in real-time systems. This predictability is a result of improving the cache's "worst

case" effective memory access time which allows the processor to operate more efficiently, thus increasing its ability to meet real-time deadlines. The problem of unpredictable caches is examined and explained using cache reference inter-miss distance (IMD), IMD frequency of occurrence, and worst case effective memory access time. Two approaches for improving cache predictability are presented and analyzed. They include modifying the cache organization and developing a prefetch architecture. By modifying parameters of the cache such as cache size, block size, cache type, and degree of associativity, cache miss behavior and the distribution of IMD values can be controlled in a predictable manner. By choosing specific cache parameter values, favorable IMD cache miss distributions can be achieved that result in lower worst case effective memory access times and faster program execution times. A prefetch architecture is developed to enhance the performance and reliability of the memory subsystem. This is done in order to avoid specific (small) IMD values by "hiding" cache misses in addition to eliminating them through cache design techniques.

The final aspect of this research involves the use of reliability theory to illustrate that techniques developed to eliminate small IMD values result in real-time systems that can meet their deadlines with a high degree of reliability. This is accomplished by estimating constant failure rates and their associated confidence

intervals for specific cache architectures. This information can then be used to calculate system reliability using specific cache design parameters.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Background

Real-time computing systems perform work in the face of time constraints generated by external events, environmental factors, or interactions with other computers. In real-time systems, the correctness of applications depends not only on the logical computation being performed, but also on the time at which results are produced. Real-time systems fall into two basic categories - *hard* and *soft*. A hard real-time system has "hard" program execution deadlines that the computer must meet or catastrophic consequences could result. Soft real-time systems also have execution deadlines, but if missed, don't result in catastrophe. Rather, the value of the result diminishes as time progresses.

Since real-time systems have fixed deadlines, it's essential for program execution times to be predictable so the computer's resources can be effectively scheduled to meet those deadlines. In addition, designers and programmers must know the execution time of individual tasks to determine if specific routines can meet the application's real-time requirements. For these reasons, hardware resources that may lead to unpredictable variations in program execution time are usually not allowed. One such resource is the hierarchical memory subsystem. Hierarchical memories are used to improve the overall performance of the system

by delivering code and data from memory faster, on average, than might otherwise be possible. Processors are generally much faster than the memory subsystem and a bottleneck is created that slows the exchange of information between memory and the processor. This bottleneck is caused by limited bandwidth resulting from the overall latency of the memory subsystem. As a result, the processor can waste precious cycles waiting for code and data to be delivered from memory. A hierarchical memory structure helps relieve this bottleneck problem by reducing the average memory latency, resulting in an overall increase in bandwidth.

An example memory hierarchy is shown in Figure 1.1 and consists of two levels - a cache (higher level) and main memory (lower level). The cache has much smaller capacity than main memory and as a result, also has a lower "hit rate," or probability of containing the required information when queried. However, it has a much faster access time than main memory and significantly improves the average performance of the memory subsystem. Main memory has a much larger capacity, but its access time is much slower than the cache's. Due to the storage capacity differences of each level, the hit rate at any level of the hierarchy is uncertain and may differ for each execution of a program or programs, thus resulting in unpredictable behavior in memory access times. While hierarchical memories are extremely efficient, their unpredictability often excludes

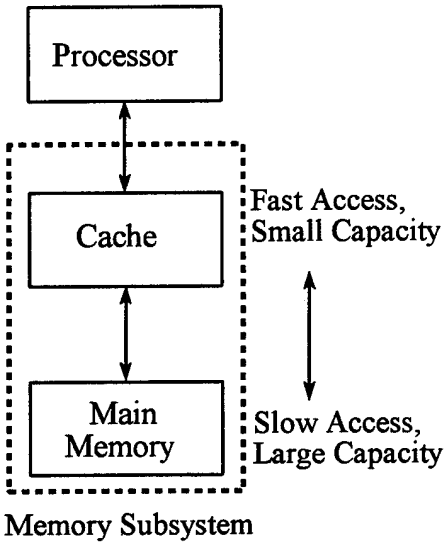


Figure 1.1 - Hierarchical Memory

them from real-time computer architectures resulting in a significant loss in overall performance. Designers of real-time systems are often forced to take the approach of designing to "worst case" specifications, frequently resulting in under-utilization of the processor. These "worst case" specifications actually lead to the most predictable program execution times which is desirable from the designer's point of view, but also lead to the worst performance in terms of program execution time. As a result, real-time systems are limited to those situations where worst case performance is acceptable. This in-turn, limits the number of potential applications that can be controlled in real-time. In addition, recent references have emphasized the need for real-time caches to support multitasking applications

[NILS94], [EGLE94].

Since reducing unpredictability while maintaining performance is the main concern for real-time system designers, the focus of recent research has been on the highest level memory subsystem - the cache. If cache memory can be used in real-time systems, hardware resources can be conserved while overall system performance is increased.

1.2 Statement of the Problem

As discussed by Koopman, the very design features that make current CPUs extremely fast can also lead to unpredictable program execution times [KOOP93]. As an example, an Interrupt Service Routine (ISR) was written and executed one million times to determine the time required for each execution of the ISR on an Intel 80486 microprocessor. This ISR was interspersed with runs of a foreground program accessing a memory array. The ISR execution time varied drastically as shown in Figure 1.2, with a "best case" execution time of 148 cycles and a "worst case" execution time of 272 cycles. This range of execution times clearly illustrates the disadvantage of using caches in real-time systems - the performance gained by caching code and data is statistical in nature, and there are no guaranteed

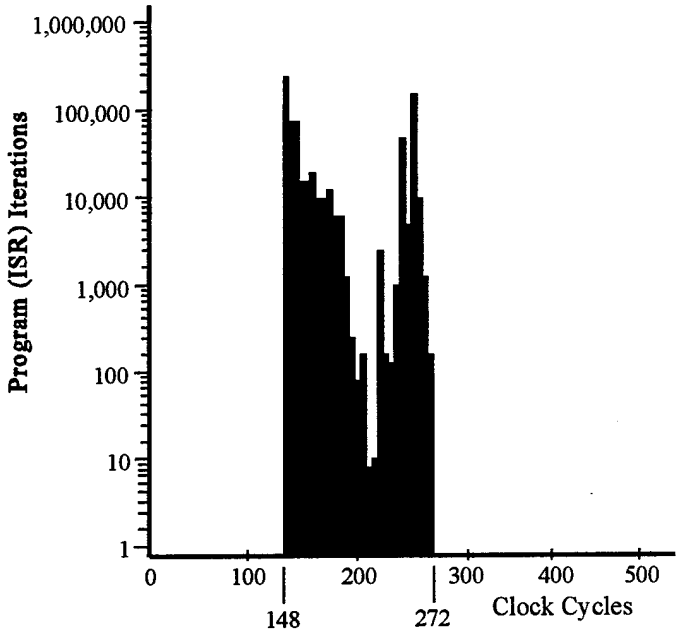


Figure 1.2 - Histogram of One Million ISR Executions [KOOP93]
(©1993 Miller Freeman, Reprinted with Permission)

execution times. Furthermore, contrary to what appears to be sharp boundaries, both the best and worst case execution times are not clearly defined. If the ISR program ran an additional one million times, there could be some ISR executions times slower than 272 cycles and some faster than 148 cycles. Since the real-time designer is limited by the worst case execution time, a conservative approach should be taken. This leads to the conclusion that the worst case execution time could be as slow as if the cache were "turned off." Figure 1.3 shows ISR program execution times with the cache both enabled and disabled, and illustrates that

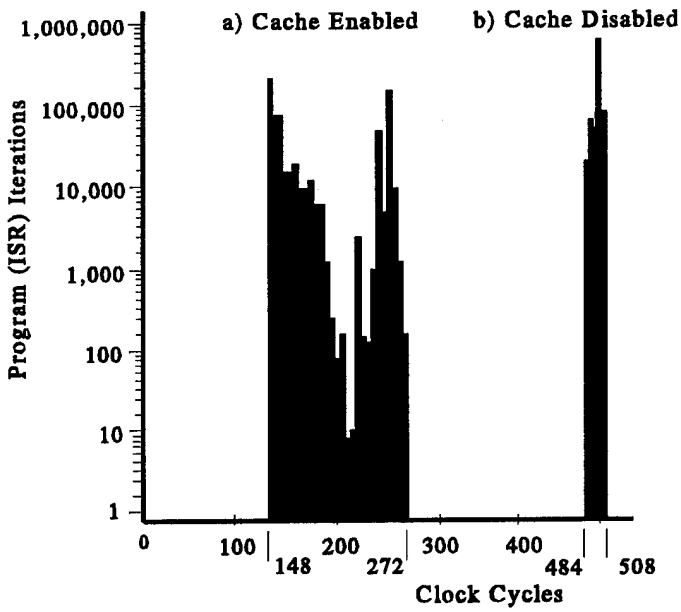


Figure 1.3 - Comparison of 1 million ISR Executions [KOOP93] (©1993 Miller Freeman, Reprinted with Permission)

execution could take as long as 508 cycles with the cache disabled (Figure1.3b).[1] Therefore the best case execution time can be considered 148 cycles while the worst case execution time must be 508 cycles. To the real-time designer or programmer, this analysis would lead to the following choices: accept the 508 cycles as worst case and use this as the design point for scheduling resources, or use something less than 508 cycles and accept the penalty associated with any

[1] Execution times vary with the cache disabled due to timing jitter associated with refreshing DRAM memory. Ideally, the execution time for this case would be constant for each individual execution of the ISR program.

missed deadlines. The first choice is often accepted by designers as the "safest" case. While this results in a design point that will cover all possible execution times, it can result in poor performance and processor under-utilization. In effect, the advantages of including cache memory are not fully realized since this is the same case as if the cache were disabled. The second choice, choosing a design point less than 508 cycles, may be acceptable for soft real-time systems, but hard real-time systems don't allow deadlines to be missed, so it is not a feasible option.

The techniques described in this dissertation significantly reduce the worst case execution time associated with the use of cache memory, thus allowing increased processor performance and utilization. By reducing the upper limit execution time (by lowering the memory subsystem's worst case effective memory access time) performance will increase in a predictable manner without any risk of missing real-time deadlines. As shown in Figure 1.4, if a design point for system timing is chosen such that it is less than the worst case, a significant risk of missed deadlines exists. The methods described in this paper eliminate this risk by reducing the worst case effective memory access time, thus allowing the processor to run more efficiently without the added risk of missing real-time deadlines.

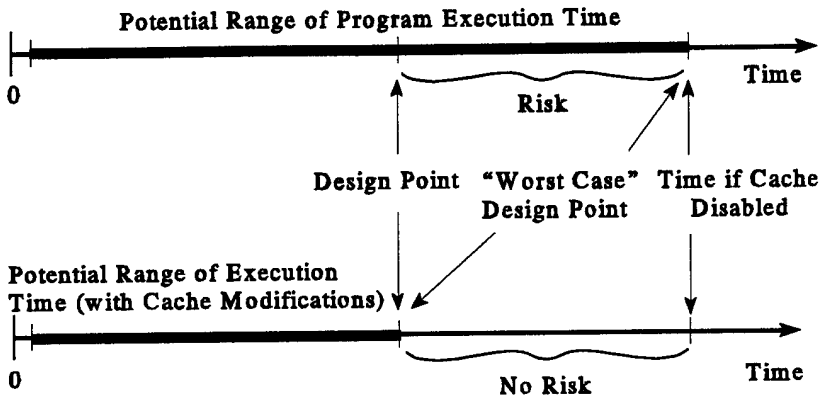


Figure 1.4 - System Design Point

1.3 Research Scope and Goals

The primary goal of this research is to develop methods that allow the use of hierarchical memory in real-time systems. These methods focus on improving the performance of the memory system's highest level component - the cache. By reducing the cache's (and memory system's) worst case effective memory access time, the resulting performance becomes more predictable. Due to more effective utilization of the processor, individual tasks are scheduled more efficiently. As a result, more events can be controlled in real-time. Since cache memories were originally developed specifically to improve processor performance for the *average case*, little work has been done in the area of improving *worst case* cache performance. However, since real-time control applications are limited by the

worst case, the research described in this dissertation focuses on improving the *worst case* timing behavior - not average behavior.

A primary method of improving cache performance is to modify the *distribution* of cache misses rather than the *frequency* of cache misses. As will be shown later, to successfully design cache memories for real-time systems, they should be designed so that misses occur infrequently with respect to one another - successive cache misses or a miss-hit-miss sequence are to avoided. To better understand this concept, Section 3.2 discusses the concept Cache Reference Inter-Miss Distance (IMD). The IMD is the "distance" between successive cache misses, and can be used as a measure of how well a particular cache design will operate in a real-time environment.

The research is divided into three parts, each summarized below and presented in the following order.

1) The first part involves examining cache architectures to determine if any relationship exists between the choice of specific cache parameters and the resulting IMD distribution, and if specific cache organizations might be helpful in optimizing worst case cache performance by eliminating small IMD values from the distribution. Parameters examined include cache type (unified or split), cache size, block (line) size, and associativity.

2) The second part of the research involves developing a prefetch architecture that facilitates prefetching code and data from main memory in an attempt to anticipate cache misses, thereby eliminating or "hiding" specific types of cache misses.

3) The third and final part of the research focuses on using reliability theory to demonstrate the usefulness of the proposed techniques in reducing effective memory access time in real-time systems. The establishment of confidence levels and intervals are used to estimate the constant failure rate and reliability of real-time cache management and design techniques.

Although a significant amount of research has been performed on real-time operating systems, this research focuses on hardware and related software techniques to manage the behavior of cache memory. Since real-time computers are typically found in embedded systems, a general assumption is made that an algorithmic scheduling policy is used to schedule hardware resource utilization and no operating system is required. Therefore real-time operating systems are not addressed in this research dissertation. In addition, the issue of real-time interrupts is not addressed. Their presence introduces a number of unique problems which are outside the scope of this research. Their study is deferred for possible future research efforts.

1.4 Organization of the Dissertation

This dissertation is organized into seven chapters and five appendices. The seven chapters describe the basis for the research, previous work accomplished, methodology, and results. The appendices provide background material for concepts used to support the research in addition to data obtained during the research.

Chapter 1 introduces background information and states the nature of the problem. An example is shown that illustrates the varying program execution times due to the use of cache memory. The scope of the research as well as goals are also discussed.

Chapter 2 discusses previous work performed in the area of memory subsystem architecture design and associated techniques to support real-time computer systems. Three basic approaches have been taken - protecting the contents of the cache by "freezing" or "locking" specific lines or blocks of the cache, protecting the contents of the cache through dynamic cache partitioning, and restoring the contents of the cache after context switches, by saving its "state."

Chapter 3 discusses how cache performance is evaluated and is presented in terms of effective memory access time, cache reference inter-miss distance (IMD), and Speedup (S). In addition, the phenomenon of cache miss reload

transients, methods used to evaluate IMD performance, and program benchmarks are discussed. These topics are introduced here and used later in Chapters 4-6 to evaluate specific cache design strategies.

Chapter 4 presents the methodology used and results of the first part of the research: modification of cache parameters and the resulting effect on IMD distribution. It also presents an analysis on various processor type - bus topology - cache type architectures to illustrate their potential impact on real-time system performance.

Chapter 5 provides the methodology and results of the second part of the research: the development of a prefetch architecture used to eliminate or "hide" cache misses causing small IMDs. Choice of prefetch algorithms, methods of implementation, and design trade-offs are discussed.

Chapter 6 presents the third part of the research: the methodology used to calculate the reliability of real-time cache architectures. The results of several simulations and the subsequent reliability calculations are given.

Chapter 7 provides concluding remarks on the research performed and discusses some areas for potential future work.

The Appendices contain detailed discussion on several supporting topics. The principles of cache associativity, prefetching methods, reliability theory,

performance evaluation and simulation tools (program tracing and cache simulation) are provided. In addition, simulation output data is also included.

Chapter 2 Previous Work

2.1 Introduction

Several approaches for making program execution time more predictable have been proposed. One approach is to not include cache memory at all, thus avoiding unpredictable behavior. Other approaches can be classified as either protection or restoration schemes. Protection implies that the contents, or portions of the contents, of the cache will be "protected" and not allowed to change while an application program is executing. Restoration schemes allow the cache's contents to change while one task is preempted by another, but is "restored" to its previous state when the preempted task is reloaded. Each approach is discussed below.

2.2 Removal of Hierarchical Memory (Cache)

For many real-time system designers, one of the most obvious choices is not to include hierarchical memories in their designs. In *Constructing Predictable Real-Time Systems*, Halang and Stoyenko state "The hardware must not introduce unpredictable delays into program execution. Hierarchical memories can lead to unpredictable variations in process execution timing. Thus caching, paging, and swapping must be either disallowed or restricted" [HALA91]. While it is clear that this choice avoids the problems associated with unpredictability, the

performance advantage of a high bandwidth, low latency memory hierarchy can't be realized. As a result, long execution times and low processor utilization may result.

2.3 Protection of Cache Contents

One approach to improving cache predictability is to "lock" or "freeze" specific blocks in the cache and protect them from being overwritten. In this scheme, the most frequently accessed routines of each task or application program are loaded into a specific area of the cache at initial program load and "dedicated" to these tasks. This code, which is generally only a portion of the entire task, is never evicted once loaded. As a result, any access to this portion of the program is guaranteed to be a cache hit. However, the number of cache hits is limited to the most frequently accessed routines. If the frequency of use drops for those routines, cache performance also drops off rapidly [KIRK90]. Figure 2.1 illustrates this technique.

Dynamic cache partitioning is another method of protection. The cache is divided into a predetermined number of partitions by the hardware. When a task is ready to run, a specific number of these partitions are requested, and all cache references are made to them. The partitions are protected during execution of the program. However, since the partitions are "owned" by specific routines, the

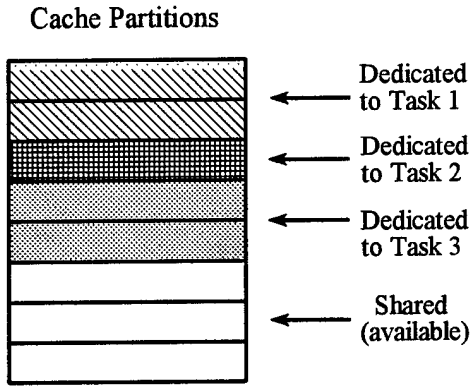


Figure 2.1 - "Freeze" Cache Blocks/Lines

possibility of insufficient cache partitions to meet outstanding requests results in some unpredictability. As a result, the success of this technique is also limited to certain applications.

Kirk [KIRK91], Wolfe [WOLF93], and Liu [LIU93] have performed recent work on optimizing cache partition schemes, resulting in improved worst case performance execution times.

2.4 Restoration of Cache Contents

Cache restoration is a technique that saves the state of the cache (or portion of the cache) for later use by a specific application, and is shown in Figure 2.2. Initially, the contents of the cache are protected until the task either completes or is preempted. If preempted, the state of the cache is saved and restored when the

task is recalled. If the task completes, the corresponding state of the cache is not saved. While this technique can work effectively, a significant amount of time is required to load the necessary routines into the cache and save the existing state. If the cache state is saved frequently, the associated overhead can significantly limit overall performance.

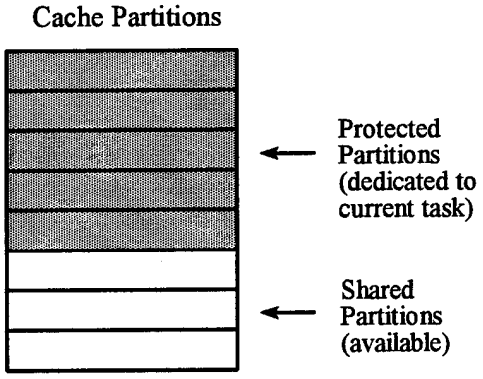


Figure 2.2 - Partitions Protected During Execution of Current Task

Chapter 3 Description of Cache Behavior

3.1 Introduction

This chapter introduces several topics that are used to describe and evaluate the behavior of cache memories designed to support real-time systems. These topics are used extensively in Chapters 4 and 5 to illustrate how modifying cache design parameters and prefetching appropriate code and data can reduce the effective memory access time and improve the predictability of cache memories. As a result, individual topics relevant to this research are introduced here, and set the stage for their later use describing specific design strategies and simulation results.

3.2 Effective Memory Access Time

As described in the introduction, a goal of this research is to develop methods and architectures that allow the use of hierarchical memory subsystems in hard real-time systems. To achieve this goal, the real-time program's worst case execution time must be reduced. This may be achieved by reducing the memory system's worst case effective memory access time, $t_{ea\text{-}wc}$.

Since program execution time is critical to real-time system design, it's useful to illustrate how the access time of the memory subsystem affects the predictability of program execution time. The access time of the memory

subsystem is that time required to load (read) or store (write) information from/to memory. Generally there are two cases - when there is a cache "hit," and when there is a cache "miss." These two cases are illustrated in Figure 3.1 and assume that an access to the cache takes one clock cycle while an access to the second level (main) memory takes ten clock cycles. The access time for a cache hit, illustrated in Figure 3.1a, is one cycle. The access time for a cache miss however, is the sum of the time required to access the cache (to determine if the information is stored there), and an access to the second level of memory to retrieve the required information. In Figure 3.1b, this totals 11 clock cycles.

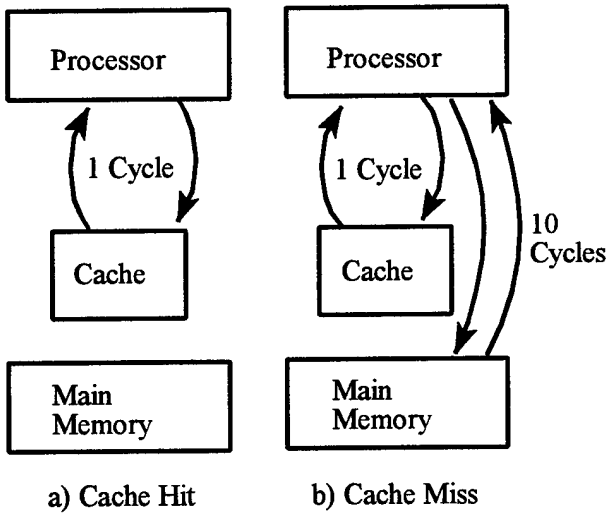


Figure 3.1 - Memory Access Time

The *effective* memory access time, t_{ea}, is the *average* time required to retrieve information from the memory subsystem and is measured over a number of cache/memory references. It can be stated as

$$t_{ea} = P_{hit} t_c + P_{miss}(t_c + T) \text{ cycles} \quad (1)$$

where P_{hit} is the weighted average probability of a cache hit, t_c is the time required to access the cache, P_{miss} is the weighted average probability of a cache miss, and T is the transport time necessary to access lower level memory (on a cache miss). Since $P_{hit} = 1 - P_{miss}$, equation (1) may be expressed as

$$\begin{aligned} t_{ea} &= (1 - P_{miss}) t_c + P_{miss}(t_c + T) \quad \text{cycles} \\ &= t_c + P_{miss} T \quad \text{cycles} \end{aligned} \quad (2)$$

The effective memory access time can be normalized to the access time of the cache, t_c leading to

$$t_{ea} = 1 + P_{miss}\, T \quad \text{cycles} \quad (3)$$

(This normalized expression for t_{ea} will be used throughout remainder of this dissertation.) The value for T may be as large as an order of magnitude greater than the time required to get information from the cache and is often referred to

as the cache's "miss penalty." For example, if $T=10$ cycles and $P_{miss}=0.1$, the resulting t_{ea} would be equal to 2 clock cycles. This leads to the conclusion that on average, the time required to access any item from the memory subsystem is 2 clock cycles. However, there may be instances where $P_{miss} \rightarrow 0$ or $P_{miss} \rightarrow 1$ resulting in access times of one and 11 clock cycles respectively. This range of memory access times obviously leads to unpredictable program execution times. In this case, the designer of a hard real-time system should assume a "worst case" access time of 11 cycles, even though the probability of it occurring may be small. This limits the number real-time applications to those that can accept a worst case t_{ea} value of 11 cycles. It is this "worst case effective memory access time", $t_{tea\text{-}wc}$, that should be reduced in order to improve processor utilization and scheduling of real-time tasks. Ideally, for a predictable real-time computer system, $t_{ea\text{-}wc}$ will always be less than the upper limit imposed by the real-time deadline.

3.3 Cache Reference Inter-Miss Distance (IMD)

In addition to the more traditional P_{miss} or P_{hit}, processor-cache performance can also be described through the use of Cache Reference Inter-Miss Distance (IMD). The IMD is the "distance," measured in cache references, between successive cache misses [VOLD81]. Example IMDs are shown in Figure 3.2.

IMD can be related to P_{miss}. For example, if IMD=3 then a cache miss

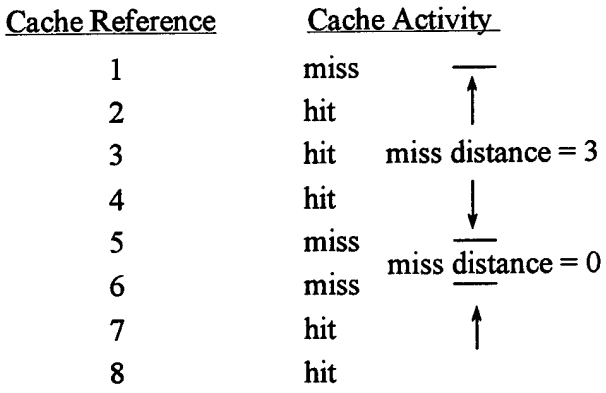


Figure 3.2 - Examples of Cache Reference Inter-Miss Distances (IMD); IMD=3, IMD=0

occurs after every third memory reference. It follows that the probability of miss can be described in terms of IMD as

$$P_{miss} = \frac{1}{IMD + 1} = \frac{1}{4} = 0.25 \qquad (4)$$

Generally, inter-miss distances vary so like P_{miss}, the IMD value would also be a weighted average of all observed IMDs. Therefore, a more accurate value of P_{miss} may be stated as

$$P_{miss} = \frac{1}{IMD_{wa} + 1} \qquad (5)$$

where IMD_{wa} is the weighted average of all IMD values. IMD_{wa} may be stated as

$$IMD_{wa} = \frac{\sum_{i=0}^{n} f_i \cdot IMD(i)}{\sum_{i=0}^{n} f_i} \tag{6}$$

where f_i is the frequency of occurrence of each individual IMD value, *IMD(i)*. However, since the real-time system performance is limited by the worst case (slowest memory access time), the "worst case IMD" is of interest. This value for IMD assumes that IMD is not a weighted average, but represents the smallest distance between consecutive cache misses (since this causes the largest delay in memory access time). It follows that the "worst case effective memory access time," $t_{ea\text{-}wc}$ is a function of the worst case (smallest) IMD value - IMD_{wc} - and can be stated as

$$t_{ea-wc} = 1 + \frac{T}{(IMD_{wc}+1)} \text{ cycles} \tag{7}$$

The two extreme values for $t_{ea\text{-}wc}$ are a result of $IMD_{wc}=0$ and $IMD_{wc}=\infty$. It follows that

$$\lim_{IMD \to 0} \left(1 + \frac{T}{IMD_{wc}+1}\right) = 1 + T \text{ cycles} \tag{8}$$

$$\lim_{IMD \to \infty} \left(1 + \frac{T}{IMD_{wc}+1}\right) = 1 \text{ cycle} \tag{9}$$

If $IMD_{wc}=0$, then all references to the memory system are cache misses ($P_{miss}=1$) and $t_{ea-wc}=1+T$ (see Figure 3.1b). This represents the "worst case." If $IMD_{wc}=\infty$, then all references to the memory system are cache hits, and $t_{ea-wc}=1$ cycle (see Figure 3.1a).

The use of IMD to describe t_{ea} provides unique information that isn't available through the use of only P_{miss} or P_{hit}. For example, if $P_{miss}=0.2$, then over ten memory references, two are cache misses and eight are cache hits. This knowledge of P_{miss} however, doesn't specify *where* the misses occur with respect to one another. IMD however, *does* specify where the misses occur in addition to their frequency. Knowing both the *distribution* and *frequency* of cache misses allows for possible solutions to the real-time cache problem that might not otherwise be possible. For example, in this dissertation, potential solutions involve the use of prefetching techniques to "hide" memory latency. This latency hiding is possible only if the distribution of cache misses is known. This is because prefetching techniques are used to hide memory latency effects during cache hits. Therefore, if a prefetching technique requires three cycles between any two cache misses to effectively hide memory latencies, the IMD should be no less than three. This approach could not be used if IMD data were not available and one had to rely strictly on P_{hit} or P_{miss} data.

It is important to note that worst case IMD values imply that the cache is experiencing misses at the IMD_{wc} rate at all times. For example, if $IMD_{wc}=0$, the assumption must be made that a miss is occurring for each cache reference, regardless of the average case (IMD_{wa}). Therefore when designing hard real-time systems, deadlines must be scheduled using t_{ea-wc} values derived from IMD_{wc} regardless of how good average IMD values may be (IMD_{wa}). This can be illustrated by the following example in which cache activity is examined while executing a program. For the first ten cache misses, if two occur on the first two references, then one miss occurs every 100 cache references thereafter, $IMD_{wc}=0$, but $IMD_{wa}=88.8$. Even though the majority of IMD values are large (IMD=100) and $IMD_{wa}=88.8$, the worst case IMD value of 0 should be used to derive real time scheduling deadlines. As a result, program execution time should be calculated using $IMD_{wc}=0$ instead of the average case ($IMD_{wa}=88.8$). This leads to the following

$$\begin{aligned} t_{ea-wc} &= 1 + T \text{ cycles} && \text{for } IMD_{wc} = 0 \\ t_{ea-wc} &= 1 + 0.011T \text{ cycles} && \text{for } IMD_{wa} = 88.8 \end{aligned} \qquad (10)$$

In essence, any substantial performance advantages gained in decreasing average program execution time by improving IMD_{wa} is wasted unless an improvement in

IMD_{wc} is achieved.

In addition to providing data on "distances" between consecutive cache misses, knowledge of IMD data also provides an opportunity to measure the performance of a specific cache architecture from a different viewpoint. In real-time systems, a goal of the system designer is to know the worst case execution time of a program or task so that additional tasks can be effectively scheduled and executed. By specifying the worst case IMD, the designer would know how frequently cache misses occur and the minimum distance between misses. From this viewpoint, efforts can focus on increasing the IMD_{wc} (making IMDs larger) as opposed to viewing the solutions as simply lowering the value of P_{miss}. Although P_{miss} and IMD are closely linked, two views of the same problem usually allow more flexibility in developing efficient solutions that may not be possible when the problem is viewed from a single angle. This additional view of cache (and memory subsystem) behavior provides the basis for the solutions presented in the remainder of this dissertation.

3.3.1 Example IMD Distribution

An example distribution of inter-miss distances, based on data published by Voldman and Hoevel of IBM is shown in Figure 3.3. A portion of their work examined the occurrence of cache misses with respect to one another and correlated

the "distance" between misses to specific causes. Voldman and Hoevel proposed that the distribution of IMDs are basically bi-modal, where the smaller IMDs are "bursts" of cache misses that occur after sudden changes in address locality (e.g., subroutine calls and context switches), while the larger IMDs are "gaps" of long duration occurring between tasks [VOLD81]. This concept is illustrated in Figure 3.4.

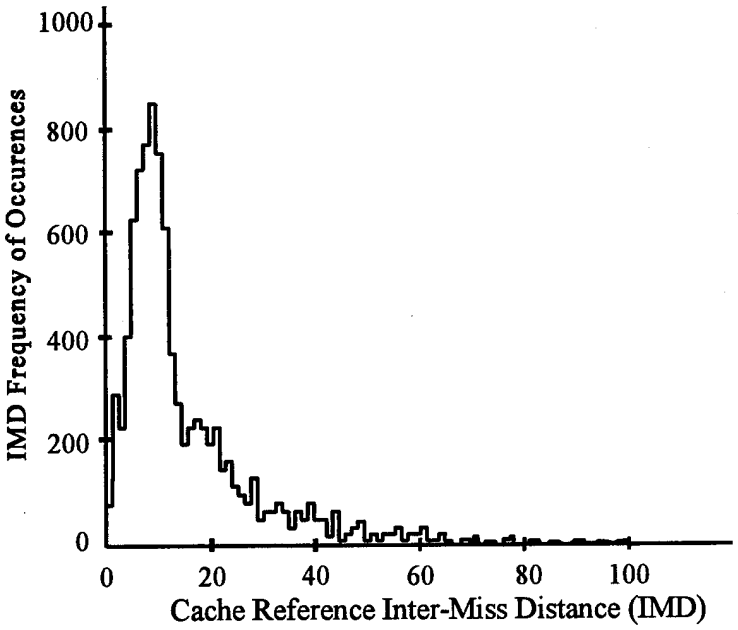


Figure 3.3 - IMD Histogram for 1 Million Memory References [VOLD81] (©1981 IBM Corp, Reprinted with Permission)

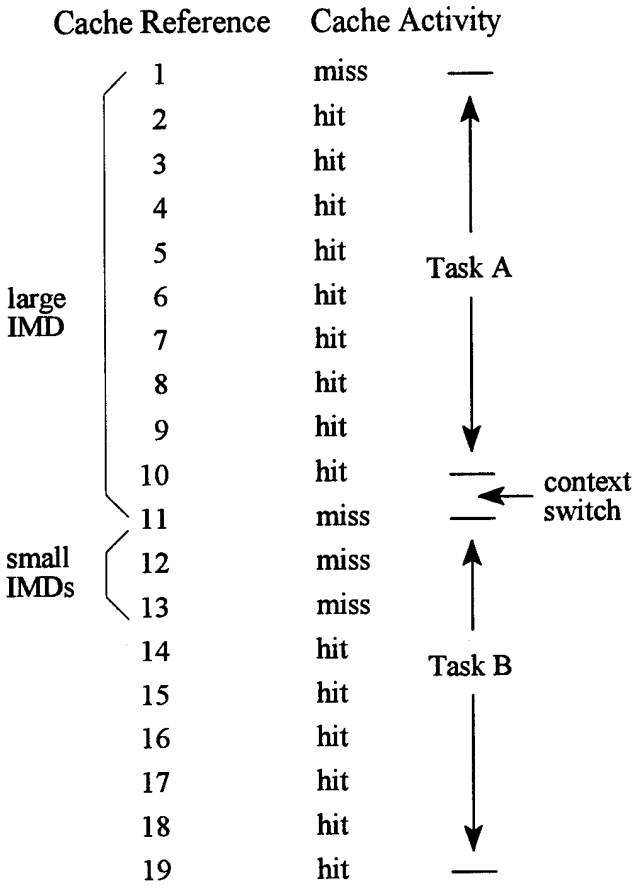


Figure 3.4 - Occurrence of IMDs due to Context Switch

3.4 Relationship Between IMD and Program Execution Time

As shown in the equation for $t_{ea\text{-}wc}$ (7), $t_{ea\text{-}wc}$ is inversely proportional to IMD_{wc}. Therefore in an effort to minimize $t_{ea\text{-}wc}$, the elimination of smaller IMD_{wc} values are of primary concern. This is because hard real-time systems are always limited by the longest program execution times possible - those caused by the largest $t_{ea\text{-}wc}$ values (and the smallest IMD values). For example, if *every* cache reference were a cache miss (worst case for program execution times, but best for predictability) then $IMD_{wc}=0$, and assuming $T=10$ cycles,

$$t_{ea\text{-}wc} = 1 + \frac{10}{0+1} = 11 \text{ cycles} \tag{11}$$

However, if every *other* reference were a cache miss, then $IMD_{wc}=1$, and

$$t_{ea\text{-}wc} = 1 + \frac{10}{1+1} = 6 \text{ cycles} \tag{12}$$

Obviously, smaller IMD values increase $t_{ea\text{-}wc}$ which will in turn, limit overall system performance. At this point, it is worthwhile visualizing how IMD values affect t_{ea}. Since t_{ea} is an *effective* time, it is calculated over a range of memory accesses including both cache hits and cache misses. It can be expressed as

$$t_{ea} = \frac{\text{total memory access time}}{\text{total memory references}} \tag{13}$$

For the cases mentions above, if $IMD_{wc}=0$, this implies that there is a cache miss for *every* memory (cache) reference. Assuming $T=10$ cycles, six cache references, and using Figures 3.1 and 3.5a as illustrations, it can be seen that

$$t_{ea-wc}(IMD_{wc}=0) = \frac{\text{total memory access time}}{\text{total memory references}} = \frac{(1+10)6}{6} = 11 \text{ cycles} \quad (14)$$

However, for $IMD_{wc}=1$, the worst case effective memory access time includes both cache hits and cache misses (Figure 3.5b) so,

$$t_{ea-wc}(IMD_{wc}=1) = \frac{3(1) + 3(1+10)}{6} = 6 \text{ cycles} \quad (15)$$

Cache Reference	Access Time	Cache Reference	Access Time
Miss	11	Hit	1
Miss	11	Miss	11
Miss	11	Hit	1
Miss	11	Miss	11
Miss	11	Hit	1
Miss	11	Miss	11

a) $IMD_{wc} = 0$ b) $IMD_{wc} = 1$

Figure 3.5 - Example of Worst Case IMDs

Since hard real-time designs are limited by the *worst case* effective memory access time, $t_{ea\text{-}wc}$ should be reduced. This requires the worst case inter-miss distance, IMD_{wc} be increased since,

$$t_{ea-wc} = 1 + \frac{T}{IMD_{wc}+1} \text{ cycles} \quad (16)$$

This $t_{ea\text{-}wc}$ will then determine the overall worst case program execution time. The worst case IMD value is determined by the smallest IMD value in any distribution. For example, if a *single* occurrence of $IMD=0$ exists in a distribution of 100,000 IMD values, then $IMD_{wc}=0$ (although the weighted average of all IMD values would be greater than zero).

The effects of IMD values on program execution time are shown in Figure 3.6. This figure shows the program execution times measured by Koopman and previously illustrated in Figure 1.2. When the cache is enabled, multiple IMD values occur and as shown, execution times vary drastically. The largest execution time, 272 cycles, is caused by the worst case (smallest) IMD value, while the fastest execution time, 148 cycles, is caused by the best case (largest) IMD value. A goal of this research can be visualized then as developing techniques that force the upper limit program execution time (272 cycles in Figure 3.6) as far to the left as possible (by reducing $t_{ea\text{-}wc}$).

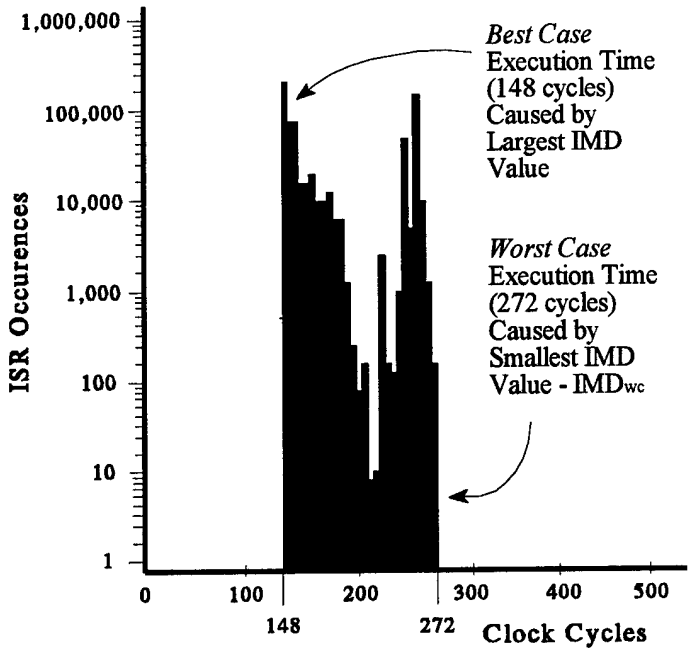


Figure 3.6 - Improve Execution Time by Eliminating Small IMD Values [KOOP93] (©1993 Miller Freeman, Reprinted with Permission)

Figure 3.7 shows the effects of eliminating subsequent worst case IMD values verses $t_{ea\text{-}wc}$. Note the drastic reduction in $t_{ea\text{-}wc}$ levels as a result of eliminating IMD values of 0, 1, and 2. Thereafter, improvements in the reduction of $t_{ea\text{-}wc}$ are less pronounced. From this point of view, improving the *distribution* rather than the *frequency* of cache misses is of primary concern. A distribution with smaller IMD values eliminated (with possibly an increased number of large IMD values) is preferred since the worst case execution time is limited by the

smaller IMD values. Research described in this dissertation focuses on eliminating *IMD*=0, 1, and 2. Elimination of *IMD*=0 is the primary concern,[2] followed by the elimination of *IMD*=1, *IMD*= 2. As shown in Figure 3.7, the elimination of small IMD values provide the greatest decrease in $t_{ea\text{-}wc}$ and will result in reduced program execution times.

It is interesting to note that while decreasing $t_{ea\text{-}wc}$ (and the subsequent program execution time), the *average* program execution time may not change a significant amount. This is because the occurrence of worst case conditions is generally an infrequent event compared to the vast majority of program executions so the value of IMD_{wa} may not change considerably. However, to the hard real-time designer, this infrequent occurrence of worst case conditions is of primary concern and should be used as the "design point" for scheduling tasks. As a result, by controlling the occurrence of specific IMD values, the real-time system enjoys improved worst case performance in addition to excellent average performance. Methods to control these IMD values are described in Chapters 4 and 5.

---

[2] This is because *IMD*=0 is the worst case, and limits overall cache performance. Once this IMD value is eliminated, then *IMD*=1 becomes the worst case, then *IMD*=2, etc.

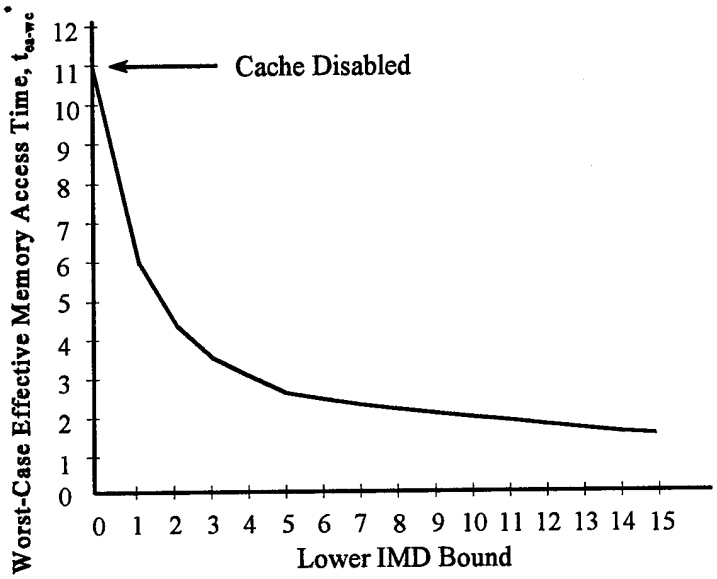


Figure 3.7 - Effective Memory Access Time vs Lower IMD Limit (* Assume T=10 cycles)

3.5 Cache States: Miss Reload Transients and Modes of Operation

During typical program execution, the cache may be viewed as being in one of two modes, or states - normal algorithm execution (steady state) where infrequent cache misses occur, or the transient state where the majority of cache misses occur. The latter state is caused by the generation of cache reload transients. These transients are characterized by several successive cache misses and are generally caused by cold starts, context switches, or other abrupt changes in memory address locality. These transients cause the generation of multiple

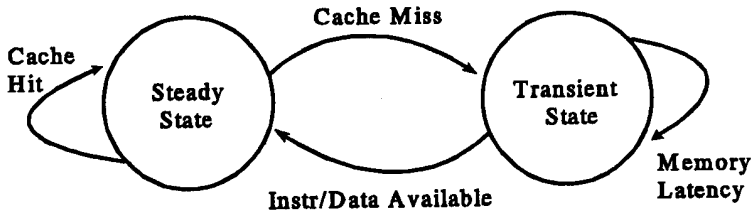


Figure 3.8 - State Diagram for Normal Cache Operation

small IMD values. Cold starts result when the cache is empty due to the start of a program (when the hardware is first initialized or following a cache flush). This situation typically causes many successive cache misses until the program and relevant data are loaded into the cache. Context switches occur when new tasks, or programs, are invoked. Until the new code and data is loaded into the cache, repeated misses will occur [THIE87]. Abrupt changes in address locality can also be caused by subroutine or function calls in address space outside of the limits currently resident in the cache. The state diagram for typical cache operation is shown in Figure 3.8. During the steady state, only large IMDs occur. Small IMD values (0, 1, etc.) are generated in the transient state. By illustrating the operation of the cache in this way, possible solutions can be examined to ensure the cache is (or appears to the processor to be) in the steady state at all times.

Using the state diagram in Figure 3.8, designing a more predictable cache can be approached in two ways. First, the behavior of the memory subsystem

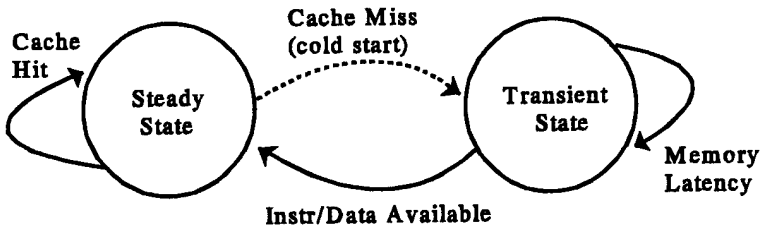


Figure 3.9 - No Return Path to Transient State

could be modified so that once in the steady state, there are no possible return paths to the transient state. After an initial transition to the transient state during program start-up, a more predictable cache would then result. This is illustrated in Figure 3.9. The second approach is to "mask" the transient state so the processor "thinks" it is always in the steady state. This can be accomplished by modifying the characteristics of the cache or by modifying the system architecture that supports the cache and processor such that the transient state is not experienced by the processor (even during program start-up). As a result, cache miss reload transients can be made "transparent" and "hidden" from the processor, resulting in a significant reduction of small IMD values. The resulting state diagram is shown in Figure 3.10.

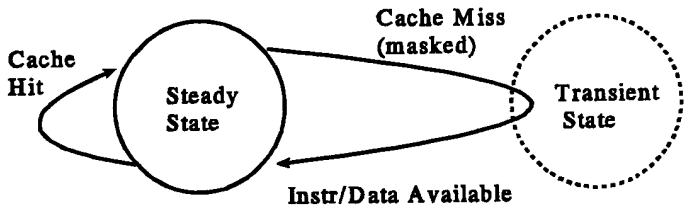


Figure 3.10 - State Diagram for Masking Transient State

Chapter 4 Modification of Cache Parameters

4.1 Introduction

The purpose of this portion of the dissertation is to develop some basic relationships between IMD behavior and cache parameters, as well as assess the effects of various design parameters on cache performance as a function of IMD. It must be noted that since cache memory performance is dynamic, its performance may differ depending on the program being executed. However, the program benchmarks used in this research are typical of those used in embedded real-time systems and accurately reflect real-time cache performance. Therefore, the general relationship established between cache parameters and IMD distribution resulting from analysis based on these program benchmarks is considered typical of embedded real-time applications.

4.2 IMD Behavior vs Modification of Cache Parameters

During this part of the research, the effect of cache organization on IMD behavior and the resulting IMD distribution is examined. Cache parameters including cache type and cache size, block (line) size, and cache associativity are varied, and subsequent IMD distributions are examined. Increasing the block size, for example, may reduce the frequency of small inter-miss distances and change the "shape" of the IMD distribution. Such a change may result in a greater

number of large IMD values and a lesser number of small IMD values. This, as previously shown, results in a smaller worst case effective memory access time, thus enhancing overall system performance by increasing processor utilization.

Initial simulation efforts focused on modifying four primary cache parameters: *cache type* (unified or split)[3], *cache size*, *block size*, and *associativity*. These parameters were chosen since they have the most dramatic impact on performance. Other parameters such as the policies for block placement (LRU, etc.), write-back/copy-back, and allocation could also be modified, but were found after some trial simulations to have only secondary effects on the cache's IMD distribution. They were set to their respective default values by the simulator (shown in the footnotes to Table D.1). Cache size was limited to 8k-128k. Smaller cache sizes were also simulated (on a limited basis), but these did not allow for satisfactory performance while caches larger than 128k did not show any significant improvement in performance. As a result, cache sizes less than 8k and larger than 128k were eliminated from further analysis. Table 4.1 summarizes the various simulated cache configurations.

---

[3] A unified cache stores both instruction and data while a split cache physically consists of two separate cache memories - one used to store instructions and the other to store data. For each simulation including a split cache, both instruction and data caches are identical in size.

Table 4.1 - Simulated Cache Parameters			
Cache Size	Cache Type	Block Size	Associativity
8k bytes	Unified and Split	16 bytes	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
		32	1, 2, 4, 8, 16, 32, 64, 128, 256
		64	1, 2, 4, 8, 16, 32, 64, 128
		128	1, 2, 4, 8, 16, 32, 64
		256	1, 2, 4, 8, 16, 32
		512	1, 2, 4, 8, 16
		1024	1, 2, 4, 8
16k bytes	Unified and Split	16 bytes	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
		32	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
		64	1, 2, 4, 8, 16, 32, 64, 128, 256
		128	1, 2, 4, 8, 16, 32, 64, 128
		256	1, 2, 4, 8, 16, 32, 64
		512	1, 2, 4, 8, 16, 32
		1024	1, 2, 4, 8, 16

Table 4.1 (continued) - Simulated Cache Parameters			
Size	Type	Block	Associativity
32k bytes	Unified and Split	16 bytes	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
		32	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
		64	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
		128	1, 2, 4, 8, 16, 32, 64, 128, 256
		256	1, 2, 4, 8, 16, 32, 64, 128
		512	1, 2, 4, 8, 16, 32, 64
		1024	1, 2, 4, 8, 16, 32
64k bytes	Unified and Split	16 bytes	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096
		32	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
		64	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
		128	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
		256	1, 2, 4, 8, 16, 32, 64, 128, 256
		512	1, 2, 4, 8, 16, 32, 64, 128
		1024	1, 2, 4, 8, 16, 32, 64
128k bytes	Unified and Split	16 bytes	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192
		32	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096
		64	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
		128	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
		256	1, 2, 4, 8, 16, 32, 64, 128, 256, 512
		512	1, 2, 4, 8, 16, 32, 64, 128, 256
		1024	1, 2, 4, 8, 16, 32, 64, 128

4.2.1 IMD Evaluation Methods

Several methods exist to evaluate the performance of cache architectures including hardware monitoring, analytical methods, and simulation [LAHA88]. For this research, trace driven simulation was chosen due to its efficiency and flexibility. Multiple cache simulations were performed and focused on modifying cache parameters and examining the resulting effects on the IMD distribution. Software was written to automate much of this process so that exhaustive simulation of possible cache configurations could be conducted efficiently. The general steps of this method are shown in Figure 4.1. Once a real-time benchmark program is compiled into binary (executable) format, an input data file is randomly generated (to ensure independent trials). The program is then executed using the data file, and a record of both instruction and data addresses are generated or "traced." The instruction and data addresses traces, together with specific cache parameters are used as inputs to the cache simulator which outputs raw cache activity and performance information. IMD data is then extracted from this output and is plotted in histogram format.

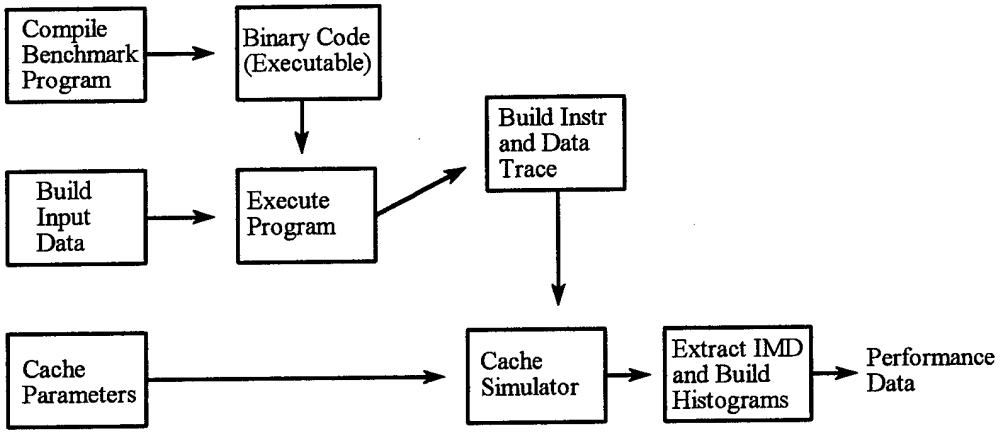


Figure 4.1 - Cache Simulation and Evaluation Process

The simulator chosen to simulate cache designs is *DineroIII* written by Mark Hill. It was selected due to its widespread use and availability [KIRK90], [LIU93]. The simulator reports the behavior of one or more alternative cache designs in response to an input program trace provided by the user and specified cache parameters.[4] Cache parameters (e.g., block size, associativity, etc.) are set with command line options [DINE94]. A unified cache (instructions and data cache together) or split cache (separate instruction and data caches) can be simulated. Additional details on *Dinero III* may be found in Section D.3 of Appendix D.

---

[4] Program traces were generated using the Quick Profiling and Tracing System (QPT), described in Section D.2 of Appendix D.

4.2.2 Program Benchmarks

To properly evaluate cache designs, two "benchmarks" programs were used to assess performance. These two programs, *SHUTTLE.c* and *LRCpr1.c*, were chosen due to several factors. First, they were both written in the *C* programming language, so they could be modified if necessary and easily ported to the workstation environment (Sun SPARC with SunOS UNIX operating system). In addition, they both were used in previous academic research studies to evaluate various aspects real-time systems [HELL84]. Finally, they were small enough in executable size (approximately 144k and 152k bytes) to represent a typical real-time program in an embedded system.

SHUTTLE.c controls a real-time decision system on the NASA Space Shuttle called the *Cryogenic Hydrogen Pressure Malfunction Procedure of the Space Shuttle Vehicle (SSV) Pressure Control System.* It is invoked in every monitor-decide cycle to diagnose the condition of the Cryogenic Hydrogen Pressure Control System and to give advice for correcting the diagnosed malfunctions. This program was slightly modified to reduce the number of "artificial" function calls. The program uses a number of *printf* statements to print results to standard output (the screen). However, when implemented, the more likely output would be some type of actuator receiving commands from an output

port of the computer. These would likely involve *OUT* assembly language commands, which wouldn't require any references to libraries (*printf* function calls). The intent is to avoid artificially introducing small IMD values. Input commands using *scanf* function calls were not modified since there was no way to inject data into the program without them. However, the likely scenario would involve the use of *IN* assembly language commands. *SHUTTLE.c* requires a binary input file of 31 lines, each consisting of a single binary digit. The pattern of binary values indicates the status of simulated cryogenic hydrogen pressure sensors aboard the SSV. Any random pattern results in an output of pressure sensor readings until a fixed point is reached. Using a random input file, a program trace was generated.

The *LRCpr1.c* program models the Local Reactivity Controller (LRC) of a nuclear reactor. The LRC controls the reaction in the reactor core by moving control rods in and out of the core. The actual movement of the rods is actuated by stepper motors which are driven by electrical pulses controlled by the LRC. An electrically activated brake is also used with each control assembly. The brake prevents the heavy control assembly from moving into the core when the stepper motor is not energized. The electrical pulses required by the stepper motors are provided by a power source and are distributed by the rod selector. The selector

ensures only one stepper motor is connected to the power source at any time. The purpose of the program is to detect single failures in each of the active pieces of equipment controlled by the LRC. *LRCpr1.c* requires an input file of 10 integers, each on a separate line. Each integer simulates sensor inputs that monitor equipment controlling the movement of control rods. Program output differs depending on whether or not the equipment status data indicates failure. Using two sets of data (indicating successful equipment status as well as equipment failure), program traces were generated.

4.2.3 Simulation Results

Each simulation run resulted in a substantial amount of data. Portions of the data were plotted in three forms: 1) the total number of all IMD occurrences versus their respective frequency (IMD distribution), 2) the number of occurrences (count) of *IMD*=0, 1, and 2 versus associativity for a fixed block size, and 3) IMD count versus block size for a fixed associativity. The IMD distribution for a typical simulation run is shown in Figure 4.2 and clearly resembles that illustrated by Volmand and Hovel (see Figure 3.3).

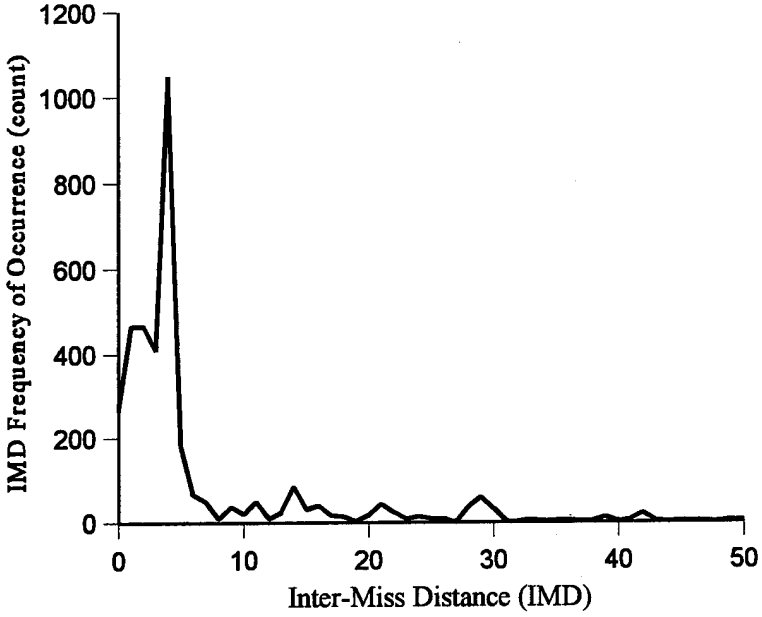


Figure 4.2 - IMD Distribution, SHUTTLE.c Benchmark, Unified Cache Size=16k Bytes, Block Size=16 Bytes, Associativity=1

Tables 4.3, 4.4, 4.5 and E.1 through E.30 list the results for *IMD*= 0, 1 and 2 count and the corresponding cache size, block size, and associativity for the *SHUTTLE.c* benchmark program. Tables E.31 through E.60 show the IMD count for the *LRCprl.c* benchmark when input data indicating successful system status is used (*LRCprl.c.ok*). Tables E.61 through E.90 show the IMD count for input data indicating system failure (*LRCprl.c.fail*). Associativity values of 1 through 16 are shown for both programs. Direct mapped (1-way associative) caches are faster than multi-way associative caches due to logic path design requirements.

Therefore, it's desirable to determine how direct mapped caches behave as a function of IMD values. However, due to less flexibility in placing blocks, direct mapped caches can force necessary blocks out of the cache resulting in "conflict" misses. Multi-way associative caches help avoid this problem, but are slower since additional hardware is required (see Appendix A). Degrees of associativity greater than 16 are not shown because the resulting IMD counts reach an asymptotic value for larger degrees of associativity.

Each of the tables represent the total count of IMD values (0, 1, and 2) for a single execution of the benchmark program, given specific cache design parameters. Table 4.2 for example, illustrates the total number of occurrences of $IMD=0$ values when executing the *SHUTTLE.c* program on a 16k byte cache. Both unified and split cache configurations are used, while varying the block size between 16 and 1024 bytes (shown vertically) and the degree of associativity between 1 and 16 (shown horizontally). The same program trace is used for each measurement, so comparisons of cache behavior under the same program execution conditions can be made. Table 4.3 is a variation of Table 4.2 with the IMD count for individual elements of a split cache (instruction and data caches) shown.

Simulation output data is provided in two "views." These correspond to how the cache memory's performance is "seen." The first view of cache

performance is that seen by the processor, and those IMD values are shown in Tables 4.2, 4.4, 4.5 and E.1 - E.15, E.31 - E.45, and E.61 - E.75 in Appendix E. The processor only sees the cache as part of the memory subsystem. The processor requests information, and the memory subsystem delivers it. The processor has no idea about the specifics of the cache - size and type, block size, and associativity. It only sees cache hits or misses. The type of cache - unified or split - is transparent to the processor. For example, if a split cache is used and the sequence of events is - instruction cache hit, data cache miss, instruction cache hit, data cache miss, instruction cache miss - the processor sees an $IMD=1$ even though the *IMD* experienced by the data cache itself is zero. The second view of cache performance is that experienced by the individual cache components. In the case just described, the instruction cache will experience $IMD=1$ while the data cache will experience $IMD=0$. These IMD values are shown in Tables 4.3, E.16 - E.30, E.46 - E.60, and E.76 - E.90.

It is clear from the data in Tables 4.2, 4.4, 4.5 (and Appendix E) that by increasing the block size and degree of associativity, the IMD count for $IMD=0$, 1, and 2 is reduced in most cases. Similarly, as cache size is increased, the corresponding IMD count is also reduced. Caution should be exercised however, as the IMD=0, 1, and 2 count begins to increase after the block size achieves a

specific value. This is particularly apparent in the case of direct-mapped caches.

For *IMD*=0 count, 1 was subtracted from the cache simulator output count due to a special case. *DineroIII* assumes an occurrence of *IMD*=0 for the first fetch from memory (on a cold start). However as defined, IMD is the distance between consecutive cache misses, so this occurrence of *IMD*=0 is disregarded.

Prior to actual simulation, it was theorized that *IMD*=0 occurrences would be more difficult to eliminate than an IMD value of 1 or 2 since *IMD*=0 occurrences are generally caused by sudden changes in address locality where code or data may not be located at sequential memory addresses. However as the data from the tables show, this is not always the case. In fact it appears for example, that while a similar decrease in IMD count occurs for *IMD*=1 and 2 as block size is increased, elimination of *IMD*=1 and 2 occurrences becomes more difficult. Also, as is the case for *IMD*=0, other than for *associativity*=1, there appears to be little difference in the *IMD*=1 and 2 count for various multi-way associative caches for large cache and block sizes. In addition, for all cases there appears to be an "asymptotic" value for both block size and associativity at which no additional reduction in *IMD*=0, 1 or 2 count occurs regardless of how much block size and degree of associativity are increased. Since eliminating *IMD*=0 results in gaining the most performance (see Figure 3.7), the relationship between the

IMD=0 value and the cache architecture is of primary concern. If the occurrence of *IMD*=0 can't be eliminated, efforts in eliminating *IMD*=1, 2, ... are fruitless. This is because *IMD*=0 is the worst case, and limits overall cache performance. Once this IMD value is eliminated, then *IMD*=1 becomes the worst case, then *IMD*=2, etc.

Table 4.2 - IMD=0 Count, 16k Unified/Split Caches, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	267	119	107	100	99	178	97	97	96	95
32	115	49	44	37	34	38	32	35	31	30
64	28	21	21	16	17	15	11	12	10	10
128	43	11	9	11	11	7	6	5	7	5
256	71	4	2	4	4	3	1	1	1	1
512	510	5	0	2	2	3	0	0	0	0
1024	2312	0	0	0	0	6	0	0	0	0

Table 4.3 - IMD=0 Count, 16k Instruction/Data Caches, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	129	85	85	83	82	112	97	97	97	97
32	26	24	24	23	22	24	10	10	10	11
64	5	4	4	4	4	18	4	4	4	4
128	3	3	3	3	3	61	4	6	3	4
256	1	1	1	1	1	90	4	3	4	3
512	0	0	0	0	0	199	3	2	2	3
1024	0	0	0	0	0	363	4	3	3	3

Table 4.4 - IMD=1 Count, 16k Unified/Split Caches, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	465	156	148	129	124	271	138	132	121	121
32	165	70	69	59	63	97	64	62	54	53
64	106	39	36	34	35	33	32	31	24	26
128	60	11	10	10	11	18	8	8	7	8
256	53	6	6	7	8	16	5	4	5	5
512	116	2	1	1	2	34	1	1	1	1
1024	208	8	2	2	2	60	1	1	1	1

Table 4.5 - IMD=2 Count, 16k Unified/Split Caches, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	466	197	192	166	166	273	171	169	158	158
32	144	78	70	65	62	100	59	57	53	53
64	45	41	35	26	27	27	21	19	20	19
128	33	34	31	11	12	20	8	8	8	9
256	20	8	7	6	8	15	5	5	5	6
512	100	14	13	7	6	18	4	4	5	5
1024	240	16	20	6	40	18	4	5	5	5

The findings of these simulations suggest that it *is possible* to eliminate specific instances of IMD values (i.e., $IMD=0$) by carefully choosing cache parameters. In eliminating such IMD values, the following general guidelines are presented:

1) Large caches (≥ 64k bytes for these benchmarks) are required.

2) Large block sizes (≥ 512 bytes for these benchmarks) are required.

3) Split caches generally perform better than unified caches, especially for direct mapped caches (*associativity* =1).

4) Multi-way cache associativity generally allows smaller cache and block sizes to be used; in addition, they offer better performance if unified caches are implemented.

5) If large, split caches are implemented (≥16k bytes), direct mapping generally provides equivalent reduction of $IMD=0$ count as multi-way associative caches. Therefore, direct mapping may be used.

6) In an attempt to eliminate IMD values other than 0, additional techniques (in addition to modifying cache parameters) are required.

When examining the performance of individual cache types (instruction and data), some interesting patterns appear. As Table 4.2 shows for the case of cache block sizes of 512 and 1024 bytes for the split cache (shaded cells in table),

IMD=0 count values can be eliminated for 2-, 4-, 8-, and 16-way associativity and greatly reduced for 1-way associativity. These IMD values are seen by the processor. However, when individual instruction and data caches are examined in Table 4.3, no *IMD*=0 values occur in the instruction cache, but several exist for the data cache.[5] This leads the following conclusions for split cache architectures:

1) While the processor sees few, if any *IMD*=0 occurrences, there are many cases of *IMD*=0 for the data cache. This indicates that data cache experiences multiple consecutive misses, each separated by at least one successful reference to the instruction cache.

2) Retrieving data from the data cache is more difficult than retrieving instructions from the instruction cache. This is clearly evident by examining the *IMD*=0 values for each simulation run. While eliminating *IMD*=0 values for the instruction cache is possible in many cases, eliminating *IMD*=0 for the data cache was not observed.

3) The principle of spatial locality favors instruction caches. This is illustrated in each of the tables listing instruction and data cache simulation results. Instructions tend to flow "in-line" while data tends to be dispersed

---

[5] Additional split cache data may be found in Appendix E.

in memory leading to a greater number of cache misses.[6]

4.3 Processor-Bus-Cache Topologies

The results of the simulations discussed in the previous section lead to some additional architectural concerns not previously anticipated. For example, although the processor "sees" few, if any $IMD=0$ values for the 16k split cache example (shaded cells of Table 4.2), many actually appear for the data cache when the instruction and data caches are examined separately (shaded cells of Table 4.3). The processor views a single bus architecture to the split cache, shared by both cache memory components. However, if the instruction and data caches are examined individually, the processor-cache connection appears as a dual bus - one bus for the instruction and another for the data cache.[7] An example is shown in Figure 4.3.

---

[6] "In-line" instructions are loaded in sequential memory locations and are read (and executed) in-order. In general, data is dispersed in memory. However, there are some program applications that exhibit excellent spatial locality for data due to extensive use of arrays.

[7] During simulation runs, the cache simulator models a single bus architecture connecting the processor and cache.

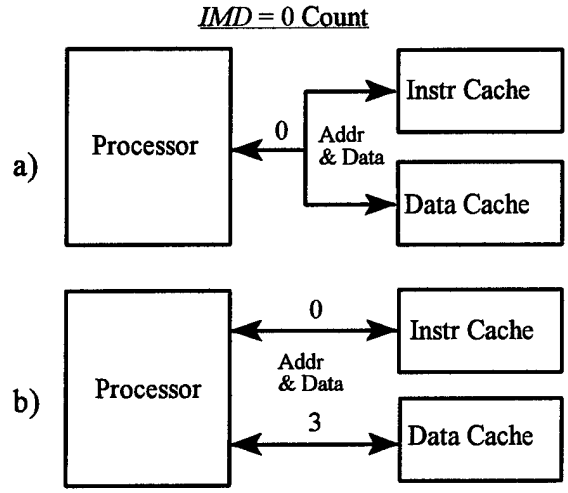


Figure 4.3 - IMD=0 Count for 16k Split Cache, Block=512, Assoc=2; a) Single Bus, b) Dual Bus

As a result of these findings, an analysis of various processor-bus-cache topologies was performed to determine their effect on system performance and to choose the topology with the most desirable worst case performance. These effects are described in Section 4.3.2 and are examined in terms of $t_{ea\text{-}wc}$, Clock Cycles Per Instruction (CPI), and Speedup, S.

4.3.1 Performance Metrics

The effects of processor type (serial or pipelined), bus type (single or dual) and cache type (unified or split) have a significant impact on performance of real-time systems. Until this point of the research, architectural performance focused

on $t_{ea\text{-}wc}$, and methods to reduce it. However, the following analysis shows that $t_{ea\text{-}wc}$ does not always fully describe the real-time system's performance. Although reducing $t_{ea\text{-}wc}$ is of primary concern to the memory system, the processor-bus-cache topology greatly influences the predictability and performance of the real-time system (and the prefetch architecture presented in Chapter 5). For example, when dealing with pipelined processors, CPI should be examined since stages of program execution may be overlapped. The pipeline processor's ability to execute pipeline stages concurrently allows significant performance advantages in terms of CPI for most cases. To evaluate the effects of cache organization and memory system architectures on performance, system speedup, S, can be used as a means of measuring enhanced performance.

CPI and S are two attractive metrics because they include the effects of processor type, bus type, and cache type. Since all possible program execution delays associated with a system's processor-bus-cache topology are included in CPI and S, they can be used as good overall measures of the system's predictability or performance. Since worst case program execution time is of primary interest, it follows that expressions derived for CPI and S should also consider the worst case: CPI_{wc} and S_{wc}.

CPI_{wc} can be defined as

$$CPI_{wc} = \frac{\textbf{maximum CPU cycles for program execution}}{\textbf{total instruction count}} \qquad (17)$$

Since CPI_{wc} depends primarily on the processor type, two expressions can be derived for CPI_{wc} - one based on a serial processor architecture, and the other for a pipelined processor architecture.

For a serial processor, there is no parallel fetching of instruction and data, and CPI is generally several cycles. It varies depending on the type of instruction executed and the instruction mix (ALU vs data references). This value for CPI can be stated as

$$CPI = \textbf{CPU cycles} + \textbf{fetch instr delay} + \textbf{fetch data delay} + \textbf{run-time delays} \qquad (18)$$

and can be simplified to

$$CPI_{wc}(\textbf{serial processor}) = n + d_{wc} + \Delta \quad \textbf{cycles per instruction} \qquad (19)$$

where n is the number of cycles associated with the various stages of the CPU, d_{wc} is the worst case delay associated with memory accesses, and Δ are any run-time delays (branch delays, delays due to dependencies, etc.). The value for n depends on the number of stages in the CPU as well as the instruction being executed, while d is dependent not only on the instruction but also the memory system. The

worst case memory delay, d_{wc}, is the sum of the time required to access instructions during an instruction fetch and the time required to load or store data.

For the pipelined processor, each stage of the processor is overlapped so instructions and data may be fetched during the same clock cycle. As a result, more than one instruction may execute at any one time. CPI for the pipelined processor case can be stated as

$$CPI = \text{steady state} + \text{memory delays} + \text{run-time delays} \quad (20)$$

and can be simplified as

$$CPI_{wc}\,(\text{pipelined processor}) = SS + d_{wc} + \Delta \ \text{cycles per instruction} \quad (21)$$

The steady state, SS, assumes the pipeline is fully loaded and there are no other delays, so SS is the minimum CPI value for that particular architecture, usually equal to 1.0 for a non-superscalar processor. Memory delays include any additional delay greater than the minimum assumed for the SS.[8] Run-time delays include those caused by branches, data dependencies, etc.

Speedup, S, is a relative measure of system performance after

---

[8] For the steady state, a memory access of one cycle is assumed, indicating a cache hit.

enhancements to a system have been made. It may be expressed as

$$S = \frac{\textbf{execution time without improvement}}{\textbf{execution time with improvement}} \qquad (22)$$

While this is useful for determining the speedup of the overall system, it assumes an execution time is known for all cases. A more useful measurement expresses S in terms of the memory system's $t_{ea\text{-}wc}$. By using $t_{ea\text{-}wc}$, S becomes a comparison of two systems - one without any modification to the memory system (baseline system) and one that includes techniques discussed in this dissertation (improved system). System speedup then becomes

$$S = \frac{t_{ea\text{-}wc} \textbf{ of baseline memory system}}{t_{ea\text{-}wc} \textbf{ of improved memory system}} \qquad (23)$$

For example, the baseline's worst case performance may assume there will be a miss on every memory reference (worst case for performance, best case for predictability) resulting in $IMD_{wc}=0$. However, if a cache memory is included along with other modifications, it may be possible to eliminate the occurrence of $IMD=0$ values, leading to $IMD_{wc}=1$ for the improved architecture. As a result, two values for $t_{ea\text{-}wc}$ can be calculated (assume $T=10$ cycles for both cases). For the baseline system (no cache)

$$t_{ea\text{-}wc} = T = 10 \text{ cycles} \qquad (24)$$

For the modified memory system (cache included)

$$t_{ea-wc} = 1 + \frac{10}{1+1} = 6 \text{ cycles} \quad (25)$$

The reduction in $t_{ea\text{-}wc}$ for the improved system is 40% less than the baseline system and results in the following speedup

$$S = \frac{10 \text{ cycles}}{6 \text{ cycles}} = 1.67 \quad (26)$$

This value for speedup illustrates how much processor utilization can be increased. The number of tasks scheduled can be increased 1.67 times without missing any real-time deadlines. Since real-time designers should plan for the worst case, and many hard real-time designs include no cache, the baseline system should assume a value of $IMD_{wc}=0$. The resulting value for speedup may be considered the "worst case" speedup, or S_{wc}. An accurate description for the worst case speedup may be expressed as

$$S_{wc} = \frac{\text{max program execution time, no enhancements (base system)}}{\text{max program execution time with enhancements}} \quad (27)$$

and may also be expressed in terms of CPI_{wc} as

$$S_{wc} = \frac{CPI_{wc} \text{ (base system)}}{CPI_{wc} \text{ (enhanced system)}} \quad (28)$$

4.3.2 Analysis of Processor-Bus-Cache Topologies

The purpose of this analysis is to determine which topology exhibits the most favorable worst case performance. For this analysis, the base cases include the serial processor and pipelined processor systems with no cache. These were chosen because they exhibit the worst case in terms of memory access time and processor utilization and represent the design of choice for many current hard real-time systems. Since worst case performance is of interest, the assumption should be made that all instructions access data in addition to the instruction reference, since these types of instructions take the longest to execute.[9] While this may be unlikely from a programming viewpoint, this assumption considers the worst case, and avoids any subjective guess as to the instruction mix of ALU and data access instructions and their respective effects on system performance.

In examining the performance tradeoffs associated with processor type, bus type, and cache type, five cases are possible. Each are listed in Table 4.6.

---

[9] Memory reference instructions such as *loads* and *stores* require two references to memory - one to fetch an instruction, and one to access the operand.

Table 4.6 - Processor-Bus-Cache Topology Configurations			
Case	Processor Type	Bus Type	Cache Type
1*	Serial	Single	No Cache
2	Serial	Single	Unified or Split
3**	Pipelined	Single	No Cache
4	Pipelined	Single	Unified or Split
5	Pipelined	Dual	Split
* Base case for the serial processor ** Base case for the pipelined processor			

Case 1

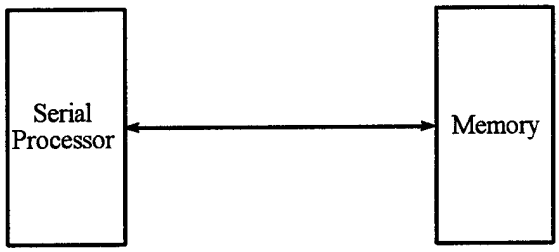


Figure 4.4 - Case 1 Topology

Since there is no cache present in this architecture,

$$t_{ea-wc} = T \quad \text{cycles} \tag{29}$$

where T is the time required to access main memory. Therefore, the memory delay, d_{wc}, is the sum of the delay to fetch an instruction and the delay to access

data during a *LOAD* or *STORE*. It follows that

$$\begin{aligned} d_{wc} &= t_{ea\text{-}wc}(\text{instruction fetch}) + t_{ea\text{-}wc}(\text{data access}) \\ &= T + T \\ &= 2T \ \text{cycles} \end{aligned} \tag{30}$$

From equation (19), CPI for this case can now be described as

$$CPI_{wc}\ (\text{Case 1}) = n + 2T + \Delta \ \text{cycles per instruction} \tag{31}$$

Because this is the base case for the serial processor, there is no speedup (S=1).

Case 2

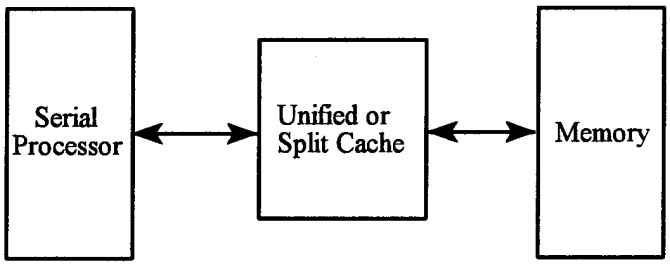


Figure 4.5 - Case 2 Topology

Since this architecture includes a cache (unified or split) $t_{ea\text{-}wc}$ is expressed as

$$t_{ea\text{-}wc} = 1 + \frac{T}{IMD_{wc} + 1} \ \text{cycles} \tag{32}$$

As in Case 1, the delay d_{wc} is the sum of the delays associated with instruction

fetches and data accesses, so

$$
\begin{aligned}
d_{wc} &= t_{ea\text{-}wc}(\text{instruction fetch}) + t_{ea\text{-}wc}(\text{data access}) \\
&= 1 + \frac{T}{IMD_{wc}+1} + 1 + \frac{T}{IMD_{wc}+1} \\
&= 2\left(1 + \frac{T}{IMD_{wc}+1}\right)T \text{ cycles}
\end{aligned}
\tag{33}
$$

Therefore,

$$
CPI_{wc}\ (\text{Case 2}) = n + 2\left(1 + \frac{T}{IMD_{wc}+1}\right) + \Delta \text{ cycles per instruction} \tag{34}
$$

The speedup for this architecture with respect to the serial base case (Case 1) is

$$
S_{wc}\ (\text{Case 2}) = \frac{CPI_{wc}\ (\text{Case 1})}{CPI_{wc}\ (\text{Case 2})} = \frac{n + 2T + \Delta}{n + 2\left(1 + \frac{T}{IMD_{wc}+1}\right) + \Delta} \tag{35}
$$

Case 3

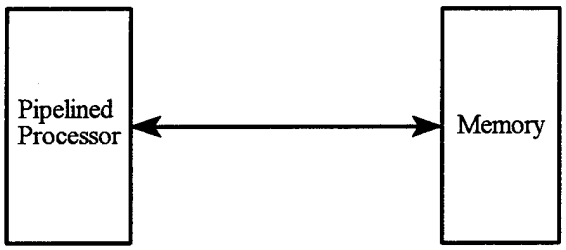


Figure 4.6 - Case 3 Topology

As in Case 1, since there is no cache present in this system, so

$$t_{ea\text{-}wc} = T \quad \text{cycles} \tag{36}$$

In addition, since there is only one bus, the IF (instruction fetch) and MEM (memory access) stages of the pipeline can't be overlapped in time. This forces the steady state of the processor to be twice the longest cycle of any stage, such as those accessing memory - IF and MEM. Since $t_{ea\text{-}wc} = T$ is the largest number of cycles (for both the IF and MEM stages of the pipeline),

$$SS = 2T \quad \text{cycles} \tag{37}$$

Since there is no cache, no additional memory delays are required, so d_{wc}=0. Therefore equation (21) can be stated as

$$CPI_{wc} \text{ (Case 3)} = 2T + \Delta \quad \text{cycles} \tag{38}$$

Because this is the base case for the pipelined processor, there is no speedup (S=1).

Case 4

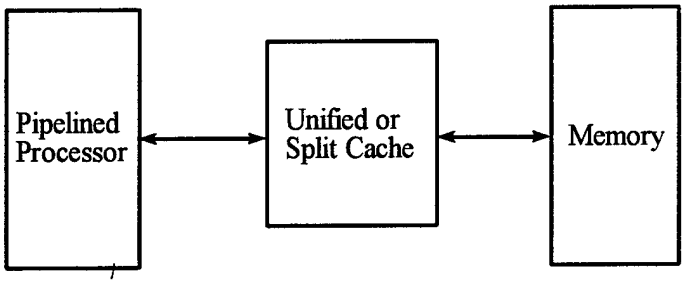


Figure 4.7 - Case 4 Topology

A cache memory is included in this architecture, so as in Case 2,

$$t_{ea\text{-}wc} = 1 + \frac{T}{IMD_{wc}+1} \quad \text{cycles} \tag{39}$$

Again, since there is only a single bus, the IF and MEM stages can't both be active at the same time, so the steady state again must be equal to twice the longest stage (IF and MEM). However, since a cache is included, steady state CPI performance assumes a cache hit for both instruction fetches and data accesses, each requiring only one cycle (instead of T cycles). Therefore the steady state performance for this architecture requires two cycles, and

$$SS = 2 \quad \text{cycles} \tag{40}$$

With the cache included, any additional memory delays, d_{wc}, can be described as the sum of delays associated with an instruction fetch and data access to main memory. Therefore,

$$d_{wc} = \frac{T}{IMD_{wc}+1} + \frac{T}{IMD_{wc}+1} \quad \text{cycles} \tag{41}$$

It follows that

$$CPI_{wc}\ (\text{Case 4}) = 2 + 2\left(\frac{T}{IMD_{wc}+1}\right) + \Delta \quad \text{cycles per instruction} \tag{42}$$

The speedup, with respect to the pipelined base case (Case 3), is

$$S_{wc}(\text{Case 4}) = \frac{\text{CPI}_{wc}(\text{Case 3})}{\text{CPI}_{wc}(\text{Case 4})} = \frac{2T + \Delta}{2 + 2\left(\frac{T}{IMD_{wc}+1}\right) + \Delta} \tag{43}$$

Case 5

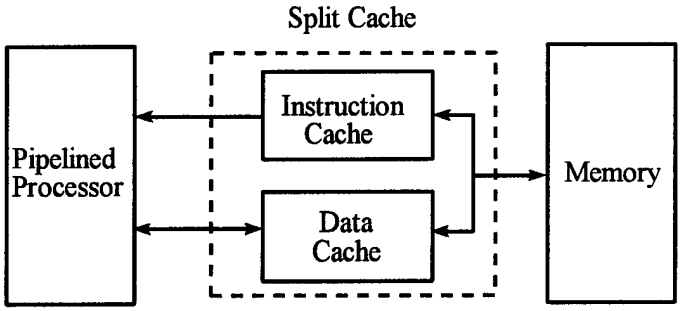


Figure 4.8 - Case 5 Topology

In this architecture, the bus architecture allows access to the instruction and data caches simultaneously, thereby overlapping the IF and MEM instruction cycles of the pipeline. This allows a steady state CPI of one clock cycle, so

$$SS = 1 \text{ cycle} \tag{44}$$

Since the IF and MEM stages overlap, the additional memory delay (required to access main memory on a cache miss), d_{wc}, will be the maximum of d_{wc} for an instruction fetch and d_{wc} for a data access,

$$\begin{aligned} d_{wc} &= \max\{d_{wc}(\text{instruction fetch}),\ d_{wc}(\text{data access})\} \\ &= \max\left\{\frac{T}{IMD_{wc}(IF)+1}, \frac{T}{IMD_{wc}(MEM)+1}\right\} \end{aligned} \tag{45}$$

It follows that

$$CPI_{wc}(\text{Case 5}) = 1 + \frac{T}{IMD_{wc}(IF/MEM) + 1} + \Delta \quad \text{cycles per instruction} \qquad (46)$$

where $IMD_{wc}(IF/MEM)$ is the lesser of IMD_{wc} for an instruction fetch and IMD_{wc} for a memory access. The speedup, with respect to the pipelined processor base case (Case 3), is then

$$S_{wc}\ (\text{Case 5}) = \frac{CPI_{wc}(\text{Case 3})}{CPI_{wc}(\text{Case 5})} = \frac{2T + \Delta}{1 + \frac{T}{IMD_{wc}(IF/MEM) + 1} + \Delta} \qquad (47)$$

To determine which of the five topologies exhibits the most favorable worst case performance, each should be examined in terms of CPI_{wc} and S_{wc}. All five cases are summarized in Table 4.7 with example CPI_{wc} and S_{wc} values for $IMD_{wc}=1$ and 2. For the serial processor architecture, the obvious choice is to include a cache memory - either unified or split - into the topology. However, for architectures including pipelined processors, the choice of topologies is not as straightforward. Cases 4 and 5 clearly provide a performance advantage over the pipelined processor base case (Case 3), but to differing degrees. The dual bus structure of Case 5 allows the IF and MEM stages of the pipeline to be overlapped, thus reducing SS to one. However, this architecture also exposes potential $IMD_{wc}=0$ occurrences for the data cache that might otherwise be hidden by

instruction fetches (between successive data accesses) for a single bus architecture. The Case 4 topology, although having a greater SS component than Case 5, is more likely to see IMD_{wc} values greater than zero,[10] and as a result will have smaller value d_{wc} components. However, as the CPI and S values illustrate, Case 5 allows faster execution time than Case 4, even for $IMD_{wc}=0$. Therefore, the Case 5 topology is chosen for the prefetch architecture presented in Chapter 5.

[10] As illustrated in Tables 4.3, E.16-E.30, E.46-E.60, and E.76-E.90.

Table 4.7 - CPI and S Comparisons for Processor-Bus-Cache Topology Cases

Case	Topology	CPI_{wc} Equation	CPI_{wc}*			S_{wc} (Serial)		S_{wc} (Pipelined)	
			IMD_{wc}			IMD_{wc}		IMD_{wc}	
			0	1	2	1	2	1	2
1**	serial processor, single bus, no cache	$n+2T+\Delta$	13.1	13.1	13.1	1	1	-	-
2	serial processor, single bus, unified/split cache	$n+2(1+\frac{T}{IMD_{wc}+1})+\Delta$	15.1	11.1	9.76	1.18	1.34	-	-
3**	pipelined processor, single bus, no cache	$2T+\Delta$	8.1	8.1	8.1	-	-	1	1
4	pipelined processor, single bus, unified/split cache	$2+\frac{2T}{IMD_{wc}+1}+\Delta$	10.1	6.1	4.76	-	-	1.33	1.70
5***	pipelined processor, dual bus, split cache	$1+\frac{T}{IMD_{wc}(IF/MEM)+1}+\Delta$	5.1	3.1	2.43	-	-	2.61	3.33

* n=5 stages, T=4 cycles, and Δ=0.1 cycles
** base case
*** IMD_{wc}(IF/MEM) is the lesser of IMD_{wc}(instruction fetch) and IMD_{wc}(data access)

Chapter 5 Prefetch Architecture Development

5.1 Introduction

Although the first stage of this research illustrates the ability to modify IMD distribution (and resulting $t_{ea\text{-}wc}$ behavior) through selection of cache design parameters, additional techniques are required to ensure the elimination of all unwanted IMD occurrences. A prefetch architecture in conjunction with carefully chosen cache parameters provides a way of "hiding" small IMD values and results in predictable cache behavior of very high reliability. This prefetch architecture prefetches required code and data from main memory and loads the cache before it is actually referenced.[11] This prefetching in effect, "hides" future cache misses and ensures the processor doesn't waste a significant number of cycles waiting for memory. This approach "masks" specific types of cache miss transients, and follows the state diagram presented in Figure 3.10. The prefetch architecture incorporates several published techniques developed by others during earlier research on instruction and data prefetching [BAER95], [CHEN95], [CONT95], [KIRK90], [KIRK91], [KLAI91], [LIU93], [McCR91], [MUEL94], [NOWI92], [SMIT78], [SMIT82], [SMIT85],[UHLI95], [WOLF93]. The research described

[11] This is in addition to the typical instruction prefetching that occurs when the instruction unit prefetches instructions from the cache and loads them in a FIFO instruction queue.

in these references is based on attempts to improve average performance, not the worst case. Since real-time systems execute a limited number of programs using *a priori* knowledge about both the programs and environment, any prefetching techniques should provide performance exceeding the average case. The added predictability afforded real-time systems will only enhance the ability of prefetching techniques in hiding cache misses, thus minimizing the worst case effective memory access time. Since this earlier research is described extensively in the stated references, development of individual prefetching techniques are not illustrated here. Rather, their used is described, accompanied by specific references to the appropriate literature.

5.2 Prefetch Architecture

A general block diagram of the prefetch architecture developed during this research is shown in Figure 5.1, and its general operation is outlined in Figure 5.2. Its basic components consist of the prefetch prediction logic (Section 5.2.1), a two-level in-line instruction queue (Section 5.2.1.2), separate instruction and data caches, and a load and store queue.[12] Since most hard real-time applications are

[12] The load queue is optional, and is based on the implementation chosen for data prefetching. It is required for hardware prefetching (Fig 5.6), but not necessary for software prefetching. A store queue is used to write data to the slower main memory and prevents any wait states from being generated.

generally limited to a specific set of tasks (application programs), an assumption is made that all possible tasks that may be executed on the particular platform are known ahead of time. This assumption of *a priori* knowledge provides the architecture's control knowledge of the program's logic path that will be executed in the near future, thus allowing prefetching of required code and data.

The prefetch architecture implements the following previously published techniques: 1) instruction and data prefetching from main memory (prediction logic and prefetch control), and 2) selective "locking" or "freezing" of cache lines/blocks to preserve necessary address space in the cache (cache control). In addition, careful selection of primary cache parameters to lower or eliminate small IMD values is also used. These three elements ensure that the information required by the current (or future) executing program is located in the cache (and queue) as needed, thereby avoiding occurrences of small IMD values. Large IMD values, caused by transients such as infrequently accessed data, are not a concern since they don't affect the worst case effective memory access time.[13] As a result, they were not treated as primary design considerations. As illustrated in Chapter 4, elimination of $IMD=0$ is possible in many cases by carefully choosing the cache

[13] Large IMD values generally don't affect $t_{ea\text{-}wc}$ since $t_{ea\text{-}wc}$ is maximized by the smallest IMD value.

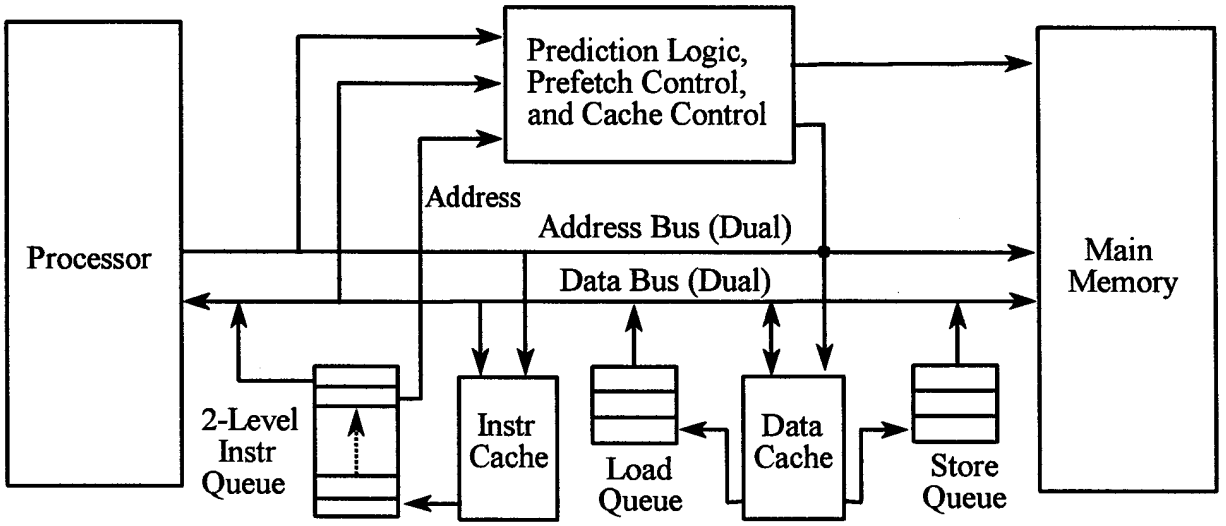


Figure 5.1 - General Block Diagram of Prefetch Architecture

size and type, block size, and associativity. However in other cases, the elimination of $IMD=0$ (and other small IMD values) are more difficult and will be addressed by incorporating instruction and data prefetching, and cache partitioning.

5.2.1 Prediction Logic and Prefetch Control

Prefetching instructions and data from main memory and loading them into a cache greatly improves the chances the required information will be available to the processor when required. In his analysis, Smith showed that by including prefetching, miss ratios were reduced 60% and 50% for instructions and data respectively [SMIT85]. This prefetching in effect "hides" or "preempts" possible cache misses by anticipating the processor's needs. As a result, these "misses" occur in advance of the processor's reference to memory. Since these prefetching actions are not seen by the processor, they are effectively "masked" from its view. The processor sees the cache as always being in the steady state (no or infrequent misses) as previously illustrated in Figure 3.10.

Since many cache miss reload transients are caused by "cold starts" (i.e., empty cache), it's imperative to pre-load portions of the cache with program code and/or data. Since the compiler has *a priori* knowledge about the programs to be executed, the first n lines of program code and data may be pre-loaded during

processor initialization but before program execution. The number of lines, n, is determined by the latency of the memory system. For example, if the memory system requires ten cycles to initially respond to the processor, then enough lines of code and to cover the ten cycle latency must be pre-loaded in to the cache. This avoids any reload transients, and ensures that the prefetch logic has enough time to initiate any prefetching actions to avoid future cache misses.

An integral part of the prefetch architecture is the prefetch algorithm that determines what and when to fetch from main memory and the method in which the algorithm is implemented. The prefetch algorithm must be carefully chosen if machine performance is to be improved rather than degraded. Too much prefetching can "pollute" the cache - prefetched lines from main memory may displace other lines from the cache which are more likely to be referenced in the immediate future [SMIT82]. It can also tie up the address and data buses unnecessarily causing the processor to wait for bus availability. Too little prefetching leads to cases where cache misses occur frequently, leading to the generation of small IMD values.

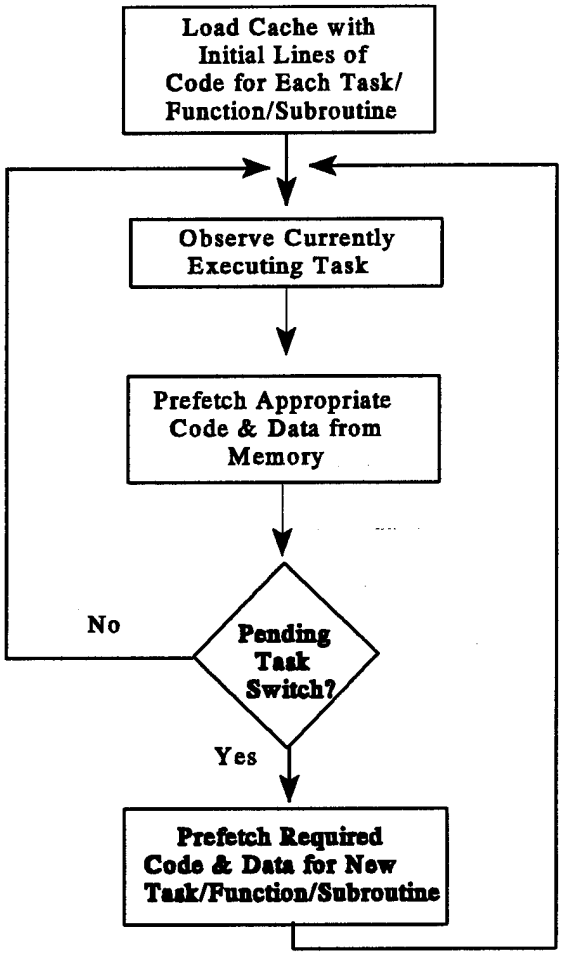


Figure 5.2 - General Flow Diagram for Prefetch Architecture (Prediction Logic and Cache Control)

5.2.1.1 Prefetch Algorithms and Methods of Implementation

There are several prefetch algorithms available, as well as methods in which to implement them in hardware and software.[14] Of the algorithms available, it appeared as a result of initial investigations that a hardware controlled implementation using the *Always Prefetch* algorithm or software controlled selective prefetching would provide the best opportunity for the reduction of $IMD=0$, 1, and 2 occurrences.

The *Always Prefetch* algorithm causes block $i+1$ to be prefetched whenever block i is referenced. This type of prefetching is simple to implement and can be very effective for prefetching sequential instructions. To determine the effect of the *Always Prefetch* algorithm on the IMD distribution, several simulations were performed that compared no prefetching (or *Fetch on Demand* (on a cache miss)) to the *Always Prefetch* algorithm for both instructions and data. It was anticipated that the number of cache misses would be reduced when the *Always Prefetch* algorithm was used, but how the distribution of those misses would be affected was unknown. A sample of the data for a 64k byte cache is shown in Table 5.1. The remaining results of this series of simulations can be found in Section B.5 in

---

[14] Prefetch algorithms and implementation methods are discussed in detail in Appendix B.

Appendix B.

Table 5.1 - IMD=0 Count, 64k Unified/Split Caches, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	103	110	91	97	90	96	90	96
32	41	45	36	37	34	36	34	35
64	17	17	11	10	10	9	10	9
128	8	11	3	3	3	2	3	2
256	4	7	0	0	0	0	0	0
512	0	9	0	0	0	0	0	0

NP: No Prefetching (Fetch on Demand), AP: Always Prefetch

As the data in the table illustrates, the implementation of the *Always Prefetch* algorithm actually degrades *IMD*=0 performance in several cases. This phenomenon is likely caused by the effects of "polluting" the cache with unnecessary blocks from memory and displacing needed blocks from the cache. For example, as illustrated in Table 5.1, if a 16 byte block is implemented, an *IMD*=0 count of 103 results when no prefetching is used and 110 occurs when the *Always Prefetch* algorithm is implemented. However, in analyzing the simulator output for cache misses (Table 5.2), the number of total misses was reduced from

953 (no prefetching) to 364 (*Always Prefetch*). This implies that the total number of misses was reduced by using the *Always Prefetch* algorithm, but those misses that did occur are clustered closer together - hence the greater *IMD*=0 count. This is likely the result of conflict misses caused by displacing needed code and data by information brought into the cache during every memory reference.

Table 5.2 - Total Cache Misses, 64k Unified/Split Caches, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	953	364	907	318	910	333	893	300
32	638	315	564	239	582	269	548	217
64	500	309	372	202	418	240	353	178
128	392	248	265	127	302	162	240	96
256	319	298	190	149	214	216	152	120
512	259	326	139	159	163	251	95	118

NP: No Prefetching (Fetch on Demand); AP: Always Prefetch

Similar findings occur for most cases: a smaller number of cache misses result when prefetching is used, but an equal or larger number of *IMD*=0 and 1 values occur. These findings support the position that prefetching via the *Always Prefetch*

or one of its derivatives (i.e., *Tagged Prefetching*) should not be used. Rather, a more selective approach should be chosen for prefetching instructions and data.

The decision not to implement the *Always Prefetch* algorithm is based primarily on the *IMD*=0 and 1 data provided by the simulator. However, architectural factors also support the decision not to use *Always Prefetch*. Although initial investigations in the literature showed some promise in using the *Always Prefetch* algorithm, the findings are based on computer architectures of non-pipelined processors and slow memory subsystems. As a result, unused machines cycles are usually available that allow additional accesses to the cache and main memory, (with the accompanying increase in traffic on the address and data buses). For example, the use of the *Always Prefetch* algorithm with a blocking cache requires two cache cycles and at least one memory cycle for each instruction or data fetch. One cache cycle is required to access the needed information and another to load the prefetched block. At least one memory access cycle is also required to load the prefetched block in addition to a memory cycle for any cache miss. The effects of these additional access to the cache and main memory on every memory reference add a significant amount of overhead to the effective memory access time and can significantly degrade performance. This is especially true with many modern pipelined processors with CPI values

approaching 1.0. Two alternative techniques are used to reduce the *IMD*=0, 1, and 2 count by selectively prefetching code and data.

The first technique is to simply increase the cache block size, thus fetching a larger number instructions and data from memory on a cache miss. As shown in Table 5.1 (and related tables in Appendix E), by increasing the block size, the occurrence of *IMD*=0 values is reduced in virtually all cases. For example, as shown in Table 5.1, by increasing the block size from 32 to 64 bytes, the *IMD*=0 count for a direct-mapped unified cache is reduced from 41 to 17. Likewise, by increasing the block size from 128 to 256 bytes for a 2-way associative implementation, the occurrence of three *IMD*=0 values are eliminated. Fetching larger blocks actually implements prefetching on a cache miss by bringing into the cache more information than necessary, thereby prefetching code and data located "close" to the information currently being used. This technique is useful for prefetching both instructions and data. This differs from the *Always Prefetch* strategy because prefetching is only initiated on a cache miss - not on every memory reference as is the case for *Always Prefetch* algorithm.

The second technique, used only for prefetching data, is based on programmer/compiler directions or by virtue of a decoded *LOAD* instruction. Two implementations are available: software and hardware controlled. Software

controlled prefetching is implemented by the compiler or programmer and initiates a prefetch only when instructed via software. This method is discussed in more detail in Sections 5.2.1.4.1 and in Section B.3 of Appendix B. The hardware implementation is that proposed by Eickemeyer and Vassiliadis and uses hardware to decode a *LOAD* instruction early in the decode cycle [EICK93]. Once identified, data prefetching is initiated automatically. This technique is discussed in more detail in Section 5.2.1.4.2.

5.2.1.2 Prefetching Instructions from Main Memory

Prefetching instructions from main memory is straightforward in most instances. Instructions are generally stored in sequential memory locations and execute "in-line" (sequentially). As a result, prefetching instructions generally requires only the knowledge of the currently executing instruction. Once that is known, the next (one or more) instructions can be fetched from sequential memory locations. Due to the spatial locality of in-line instructions, prefetching by increasing block size is best suited for prefetching instructions in this architecture. When an instruction is fetched from memory on a cache miss, the next one or more instructions are also (pre-) fetched. The choice of ideal block size is a function of the IMD distribution, and is selected for the elimination of $IMD=0$ occurrences.

While prefetching through increased block size works very well for in-line instructions, it is ineffective for prefetching instructions that may be located outside the address boundaries of the instruction space accessed by currently executing instructions. This may occur as the result of calling functions or subroutines. Subsequently, the required instructions are not found in-line and incorrect instructions may be prefetched from main memory. Traditionally, this problem has been addressed through the use of a "branch target buffer/queue" or some other method of fetching target instructions and placing them in a secondary instruction cache or queue. However, this technique requires that the transfer in program control be experienced at least once, so the relevant program data may be loaded into the buffer or queue based on historical information about the branch or subroutine. In addition, these methods are based on statistical analysis and don't guarantee cache behavior. As a result, they may not be attractive to the real-time designer. Predictable program execution time requires that the target code be as accessible as the in-line code (in terms of access time). To circumvent this problem, the first n lines of the branch target instruction stream (function or subroutine calls) are loaded into the cache where they will be available for immediate use, much like in-line code would be. This avoids any $IMD=0$ occurrences associated with the start of a function call or branch routine. The in-

line instructions executed immediately after returning to the main program from the target code are also pre-loaded to avoid a "cold-start" of the cache that could result from needed information being displaced by target code or data.[15] This can be accomplished *a priori* by using the compiler to identify the required branch target code and in-line code following the target call. Prefetching of subsequent instructions required for target routines will occur as described above (prefetch via a larger block size) once the initial target code is executed.

5.2.1.3 Prefetching Instructions from the Instruction Cache

To enhance the performance of the processor, an "In-Line Instruction Queue" is placed between the processor and the instruction cache. Its purpose is to store instructions "in-order," allowing the processor immediate access to them. The queue works on a First-In-First-Out (FIFO) basis, meaning that the instructions loaded into the queue first are also the first to exit. In addition to lining up instructions in-order for the processor, the queue also "aligns" instructions that may otherwise be "misaligned" in the cache. Misalignment may occur when the stored instructions (in the cache) do not begin on those byte boundaries accessed by the processor. For example, as shown in Figure 5.3a, if

---

[15] The number of instructions n, is determined by the maximum latency of the memory subsystem. There should be enough instructions to cover any latency such that the processor doesn't experience any wait states.

a 2-byte instruction is stored in the cache, it may be split and reside in more than one addressable block. As a result, more than one access may be required to read each instruction. However as shown in Figure 5.3b, once in the queue, instructions are aligned and only one access is required.

The in-line instruction queue used in this architecture is a two-level decoded instruction queue. It is actually a series of two queues - one that holds instructions loaded directly from the cache, and a second that holds the output of the instruction decoder (a decoded instruction). This "two-level" queue, shown in Figure 5.4, provides the control unit of the processor access to decoded instructions awaiting execution. This allows limited preprocessing which is useful for determining the status and address of the operands [MILL95]. This knowledge is used to prefetch appropriate data from the cache or main memory, or to predict any conditional branches. This queue incorporates the *Greedy Prefetch* algorithm (described in Appendix B) to keep the queue as full as possible with instructions from the cache. For this algorithm to work effectively, the state of the queue should be monitored since the rate of prefetching is not regulated or necessarily synchronized with the rate of consumption. A state diagram illustrating the prefetch algorithm for

prefetching from the instruction cache and loading the instruction queue is shown in Figure 5.5.

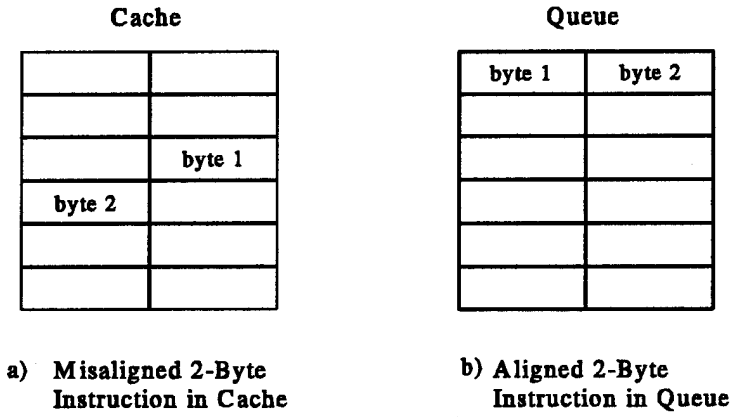


Figure 5.3 - Instruction Alignment [MILL95] (©1995 IEEE, Reprinted with Permission)

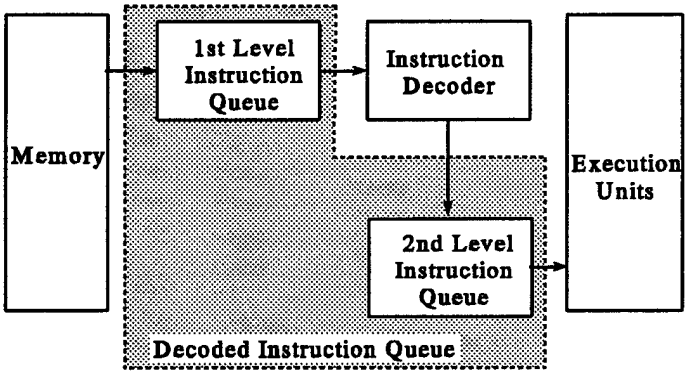


Figure 5.4 - Decoded (2-Level) Instruction Queue [MILL95] (©1995 IEEE, Reprinted with Permission)

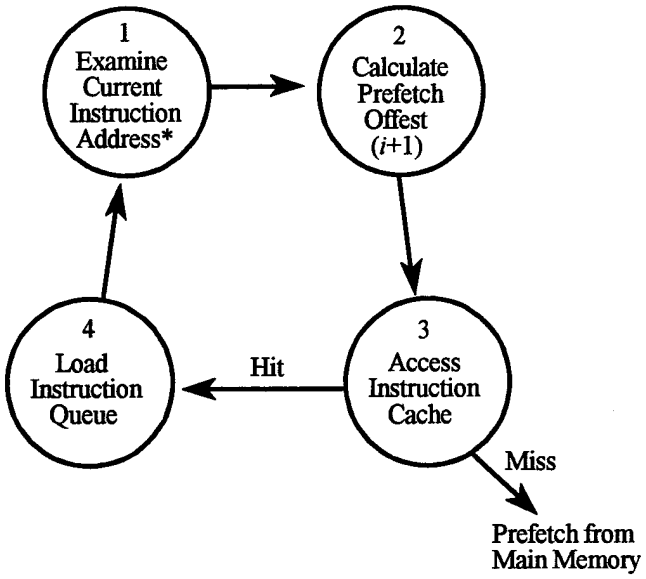


Figure 5.5 - Prefetch from Instruction Cache into Instruction Queue (*In 2-Level Queue)

5.2.1.4 Prefetching Data from Main Memory

Prefetching data is different than prefetching instructions due to some unique limitations. As previously discussed, the majority of $IMD=0$ values in a split cache are caused by misses to the data cache. This is primarily due to the fact that in general, data is not as spatially located in main memory as instructions (although some related data items such as variables and arrays are usually stored together). Therefore, there tends to be more cache misses (closer together) for data references that for instruction fetches.

The prefetching of data from main memory is the consequence of a *LOAD*

instruction decoded early in the instruction decode cycle.[16] Data prefetching must wait until this time since the data address isn't known until after the opcode is initially decoded.[17] This is in contrast to in-line instructions where the address of the next instruction is generally the next word in memory. Data is then read from memory and placed in the data cache. While this prefetching of data avoids possible cache misses, it also presents potential problems. For example, there is the possibility that the address of the prefetched data may change after the prefetch is initiated. This may occur when an instruction modifies a register that is used later by another instruction to form the data address. This is called the Address Generation Interlock (AGI) problem and can result in erroneous data being used during program execution.

Data prefetching is found in several different forms, both in software and hardware. In the software approach, prefetch instructions are inserted early in the instruction code sequence to initiate a data prefetch. The programmer or compiler identifies when *LOAD* instructions occur, and then inserts the appropriate prefetch

---

[16] *STORE* instructions are not addressed by this technique, since their operands are not prefetched. Data store operations are interfaced to memory using a store queue.

[17] As previously discussed, two-level instruction queues hold partially decoded instructions which allows operand addresses to be accessed in most cases before they are required.

instruction (*FETCH*). This approach is much more flexible than a hardware implementation since it is programmer controlled, and allows the designer possible optimization of prefetch strategies. It can be especially helpful when modifying the prefetch strategy for data with different degrees of spatial locality (i.e., arrays, variables, etc.). A number of existing microprocessor architectures have prefetch instructions as part of their instruction sets that allow programmers to specify a data fetch in advance (prefetch). Some examples are described in Section 5.2.1.4.1.

Hardware controlled data prefetching uses hard-wired logic to determine when to initiate a prefetch from memory, and can be faster than the software approach since no additional instructions are executed. Because of the difficulty of correctly determining the operand's address in advance, and after examining the significant amount of work performed in this area [ABRA93], [CHEN95], [EICK93], [KLAI91], [SMIT78], [SMIT82], [SMIT85], [STAE93] a simple, modified Always Prefetch algorithm is chosen. It fetches data on every *LOAD* instruction, but must wait for the instruction to be decoded before accessing any data. This algorithm uses the operand address information available to the processor at the time the instruction is decoded (in the 2-level instruction queue) to form the data address and initiate the data fetch. This algorithm works correctly

for all cases except those affected by AGIs. Although AGIs are a potential problem, they are not anticipated to generate a significant number of small IMDs. As shown in earlier simulations, eliminating $IMD=0$ values (as seen by the processor) is possible even if several consecutive misses are incurred by the data cache, since most data cache accesses will be separated by instruction cache accesses (instruction fetches), thereby ensuring $IMD>0$. In effect, the small IMD values generated by AGIs may be hidden between instruction fetches. For those AGIs not hidden, additional time will be required to access the necessary data, but this effect is included in CPI_{wc} (see Section 4.3.2).

5.2.1.4.1 Software Controlled Data Prefetching

Since data prefetching from main memory can be controlled in software by the programmer, examples of such cache management instructions are worth noting. Two instruction sets that support data prefetching include the Motorola/IBM PowerPC and the DEC Alpha instruction sets.

The PowerPC includes an instruction that allows a programmer to prefetch blocks of data from main memory and load them into the data cache. The *Data Cache Block Touch* instruction brings data into the cache, providing the effective address is contained in the virtual memory system's Translation Look-Aside Buffer (TLB). If there is a miss in the TLB, then this instruction is treated as a *no-op* (no

operation executed) and no prefetch is initiated. The syntax for this instruction is

dcbt rA, rB

where the effective address of the data is the sum of the **A** and **B** registers [POWE93].

Digital Equipment Corporation provides two instructions for prefetching data from memory in the Alpha microprocessor. *Prefetch Data Hint*, and *Prefetch Data - Modify Hint* each prefetches 512-byte blocks of data in anticipation of their use. They differ in that the latter instruction provides the additional hint that modifications (stores) to some or all of the data is anticipated [ALPH94]. Their syntax is

FETCH

FETCH_M

5.2.1.4.2 Hardware Controlled Data Prefetching

A basic data load unit proposed by Eickemeyer and Vassiliadis [EICK93] may be used to predict and prefetch data from the data cache for this architecture. Its block diagram is illustrated in Figure 5.6. *LOAD* instructions are detected in the second level of the two-level instruction queue before they are passed to the

pipeline execution unit. The address of the data is predicted [18] and the data is fetched from either the cache or main memory (on a data cache miss). The data is then loaded into a FIFO load queue until the *LOAD* instruction advances in the pipeline to the point when the *LOAD* normally computes the address of the data. If the predicted address is correct, then the load queue immediately supplies data to the execution unit. If the predicted address is incorrect, then another fetch from the cache or main memory must be initiated based on the correct operand address. A state diagram illustrating the data prefetch algorithm is shown in Figure 5.7.

[18] Using operand address information available to the prefetch unit at the time of prediction.

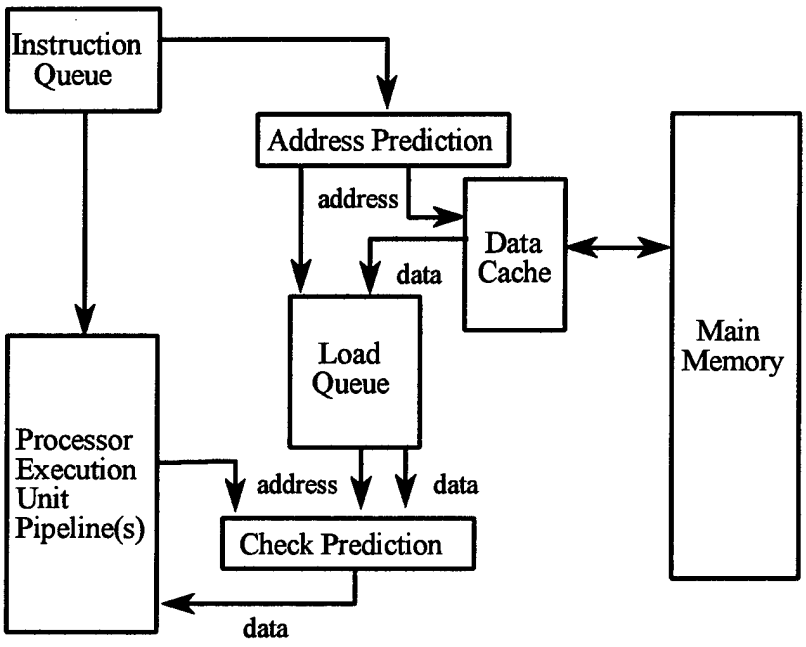


Figure 5.6 - Basic Load Unit [EICK93]
(©1993 IBM Corp, Reprinted with Permission)

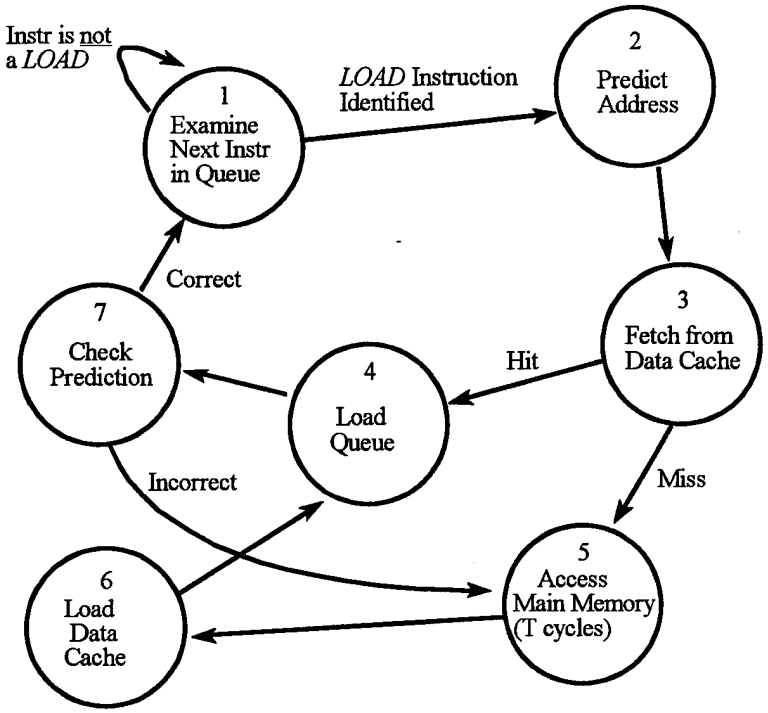


Figure 5.7 - Prefetch Data from Main Memory into Data Cache and Data Queue

5.2.2 Cache Partitioning (Locking/Freezing)

As described in Chapter 2, portions of the cache can be protected by "freezing" or "locking" individual lines or blocks in the cache. This protection is required in order to ensure that needed information won't be displaced by other instructions or data, thus removing potentially useful information from the cache. The partitioning scheme used in this prefetch architecture is called "N-Way Partitioning" (NWP), and divides the cache into individual partitions [KIRK89].

These partitions can be statically allocated to various tasks (programs) or portions of tasks. A task is forced to use only the partition which it owns, and therefore all other cache data is protected. Since portions of each partition are loaded *a priori* (before program execution), no cache miss reload transients are experienced between tasks. The cache partitioning is accomplished by mapping the cache address to a specific area of the cache defined by a "Task ID." Figure 5.8 shows an example of *Task 2* being statically bound to one of n individual cache partitions (Partition 2). This NWP scheme is easy to implement, requires only a hashing function, and provides complete protection of task specific data [KIRK89].

A Static-Locked Partitioning (SLP) approach, discussed in [KIRK90] divides an associative cache into two general areas - a smaller area for protected partitions, and a larger area for shared use - each partition is available to the task under execution. The smaller portion of the cache is divided into a number of protected partitions which are task specific and loaded with the first several lines of program code and data (main program, branch target code, subroutines, etc.) determined at compile time. Necessary library functions are also loaded into partitions in this area of the cache. Since the contents of these partitions are known, access to any information in these areas is clearly predictable. The larger portion of the cache is shared among tasks and is loaded with in-line instructions

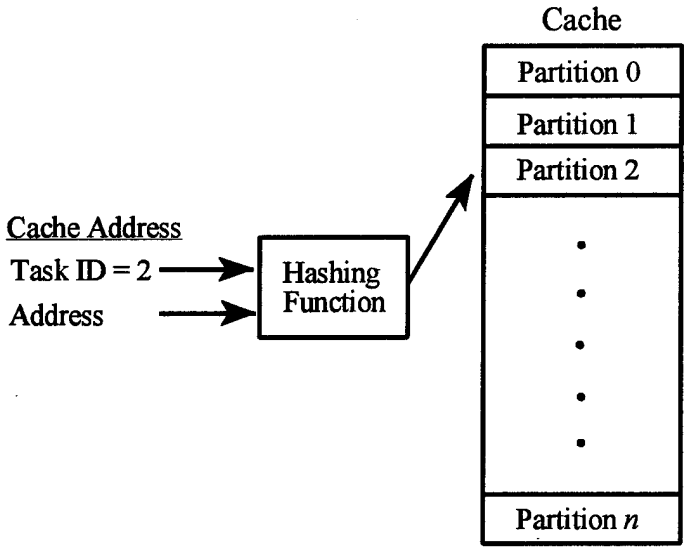


Figure 5.8 - Mapping Protected Cache Partitions
[KIRK89] (©1989 IEEE, Reprinted with Permission)

or prefetched data. Access to the protected and shared areas of the cache are determined by a mapping function which controls a Protected Area Flag (*PF*). If the flag is cleared, $PF=0$, and the shared area is accessed. However, if $PF=1$, then only the task specific partition (determined by the Task ID) may be addressed. An illustration of the partitioned cache is shown in Figure 5.9 and is similar to that used by Kirk in his Ph.D. research [KIRK90]. Both instruction and data caches are implemented in this manner. However, the amount of data pre-loaded at processor initialization may differ from the number of pre-loaded instructions.

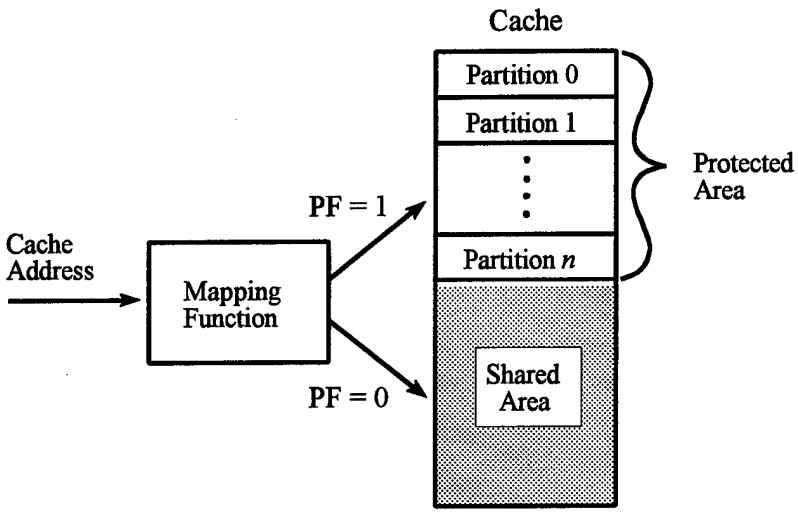


Figure 5.9 - Partitioned Cache [KIRK89]
(©1989 IEEE, Reprinted with Permission)

Direction to use either the protected or shared cache space may be embedded in the execution code as additional in-line instructions (vertical insertion) or as additional bits in the opcode (horizontal insertion). Since any access to the cache would require guidance on which partition to access, vertical insertion implies an additional instruction for each memory reference (cache access). It appears this could curtail performance, since each reference would require two instructions - a *SET/CLEAR PF* instruction and a *LOAD/STORE*. This may be significant since a large number of instructions reference memory. These additional instructions would be predictable, and known at the compile time. Any

additional execution time would also be predictable. However, since the *SET/CLEAR PF* instruction need only toggle the protection flag, only one *SET PF* or *CLEAR PF* instruction would be required for each block of protected or unprotected information. This could significantly reduce the number of PF instructions required, since individual lines of sequentially executing code and associated data will generally be in either one of the two areas. An additional advantage of using vertical insertion is that other than the mapping hardware, no architectural changes are required. Horizontal insertion, while slightly faster (since there are no additional instructions to execute) complicates instruction decoding and execution by adding additional bits to each instruction. This in-turn, requires additional data lines, wider memory components, etc.

Chapter 6 Real-Time Reliability Measurements

6.1 Introduction

The proof-of-concept used for this research is similar to that used for reliability analysis. Reliability is used to measure the probability of operational success. It can be defined as the probability that the device will perform its intended function for at least a specified period of time under certain environmental conditions. Reliability theory is used in numerous applications such as manufacturing, military systems, automotive testing, and software design. For this research, reliability theory is used to project how well the techniques and architectural modifications presented in this dissertation support the operation of real-time computer systems.

Methods for estimating failure rates for various cache designs and evaluating the reliability of real-time systems are described. These methods apply rigor to the intuitive conclusion that the more observations one makes of a system under test without failure, the more likely the system will not fail when called upon to perform its task. To estimate failure rates and reliability of real-time system designs, theory developed for predicting the reliability of physical devices is applied. A method of estimating statistical quantities, based on work previously proposed by Clopper and Pearson, is used to determine failure rate limits for given

levels of confidence [CLOP34]. Once these limits are determined, subsequent reliability values are calculated. In addition, the use of this theory is based on the assumption that the failure of real-time systems in meeting their deadlines is a "random event." This assumption may be made since all input data for each individual execution of the program is randomly generated, similar to a typical operating environment where input to the real-time system would also be randomly generated.

Since it is impossible to prove that any hardware or software system design is 100% reliable, it is likewise impossible to prove that the solutions presented in Chapters 4 and 5 will meet all real-time requirements 100% of the time. The best that can be accomplished is that the solutions are shown to meet all real-time requirements for the vast majority of the time with the probability of failure acceptably small. In the case of real-time systems, a "failure" can be defined as the case where hard real-time deadlines are not met. Thus measuring the reliability of techniques developed during this research may be measured by their success in meeting all hard real-time deadlines. This reliability is a function of the time over which reliability is measured and the constant failure rate, and is usually expressed as Mean Time To Failure, or MTTF. For this research, the number of program executions (runs) between failures or Mean Program Runs Between Failures

(MPBF) may be substituted for MTTF. The reliability is measured over the total number of individual program executions instead of the more traditional unit of measurement, time. To estimate the constant failure rate, $\bar{\lambda}$, the use of probability theory is required. Since the success or failure of a real-time system can be classified as a binary result (meet or not meet real-time deadlines), each execution of the program is either a success or failure.[19]

6.2 Failure Rates and Reliability

The failure process can be quite complex, and is often difficult to describe mathematically. As a result, a failure distribution is used to provide a statistical summary of the life over which reliability can be measured. The failure rate, is the rate at which failures occur over a certain time interval (or number of program executions), and is defined as the probability that a failure per unit time occurs in the interval, given that a failure has not occurred prior to the beginning of the interval. The hazard rate indicates the change in the failure rate over the lifetime of a population. A typical hazard rate curve for physical devices is shown in Figure 6.1. Three distinct failure regions are indicated. The first, called the

[19] Detailed discussion on reliability theory related to this research is in located in Appendix C.

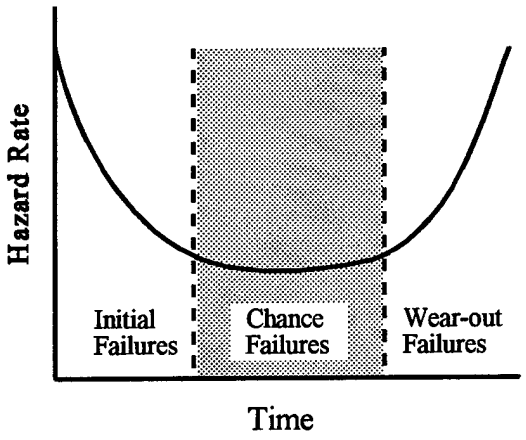


Figure 6.1 - Typical Hazard Rate Curve [MART82] (©1982 John Wiley & Sons, Reprinted with Permission)

"initial failure region" is characterized by a decreasing failure rate, and represents early failures due to material or manufacturing defects. The second region, called the "chance" or "random failure region" is characterized by a near constant failure rate. It represents chance failures caused by sudden changes in the environment. Elimination of these failures require a device that is "over-designed" for its intended environment. The third region, called the "wear-out failure region," is typified by an increasing failure rate resulting from equipment deterioration [MART82].

For the work presented in this dissertation, the design properties of cache

memories are not in themselves considered physical devices, so describing them in terms of "manufacturing defects" or "wear-out failures" does not apply. The region of interest however, is not the first or third regions, but rather the second region - the region of "chance" failures. This region shows a near constant failure rate - the same rate that would result from chance failures due to random execution of any real-time program. Most complex reliability models assume that only random component failures need be considered, thus interest focuses only on the chance failure region of Figure 6.1.

To determine the reliability of real-time systems given a constant failure rate, an exponential distribution describing reliability may be used. It can be shown that the exponential distribution accurately describes the failure time distribution of the chance failure region and is uniquely associated with a constant failure rate [MART82]. As a result, the reliability of such systems may be described as

$$R(t) = e^{-\lambda t} \tag{53}$$

where t is the time, or in this case the number of program executions, over which reliability R is measured, and λ is the constant failure rate, where $\lambda = \frac{1}{MPBF}$.

6.3 Confidence Levels, Intervals, and Limits

As an aid in determining the reliability of specific real-time systems, a level

of confidence may be given to the corresponding reliability value. This "confidence level" is given for a range of probable outcomes, and is a numerical value - usually between 0.90 and 1.0 - that is associated with the constant failure rate, and subsequently the reliability. The range of probable outcomes is called the "confidence interval" and is specified as the interval between two "confidence limits." These limits can be considered the best and worst case probabilities of success based on the initial constant failure rate estimate, $\bar{\lambda}$.

For cache memory architectures, the constant failure rate estimate may be obtained by observing failures to meet real-time deadlines for a number of program executions. For example, if 1000 program runs are executed and four failures are observed, $\bar{\lambda} = 4 \times 10^{-3}$. In the case where no failures are observed, $\bar{\lambda} = 0$. In either case, the resulting failure rate estimate may have a "confidence level" associated with it to improve its usefulness. If $\bar{\lambda}$ is based on a random sample (quantity *n)* of program executions, then it can be said that with a certain level of confidence, the true value of λ will fall within the range of confidence limits λ_1 and λ_2 such that $\lambda_1 \leq \lambda \leq \lambda_2$. Once these confidence limits are known, reliability values can be calculated at each limit, giving a range of possible reliability values. The underlying premise that allows a confidence level to be associated with failure

rates (and reliability values) is based on the statistical experience that the more often λ lies in the interval $\lambda_1 \le \lambda \le \lambda_2$, the higher confidence one can have that it will also lie within that interval in the future. In addition, the more Bernoulli trial observations are collected, the more confident one can be about the estimate and, as a result, the smaller the interval needed to assure a given level of confidence [HARN82].

6.3.1 Construction of Confidence Intervals

Clopper and Pearson present methods that allow confidence levels to be assigned to binomial probabilities [CLOP34]. A constant failure rate λ can be observed, and can be said to fall into the interval $\lambda_1 \le \lambda \le \lambda_2$ with a certain degree of confidence. The confidence interval limits, λ_1 and λ_2, are determined by calculating the region defined by λ_1 and λ_2. Constructing a 95% confidence interval implies 95% of all possible favorable outcomes calculated by the binomial equation must be included within that region. The resulting confidence interval is a range of binomial probabilities that is calculated for the constant probability of failure λ, and the number of independent trials over which λ is observed, n.

Using the binomial equation, the probability of failure in n trials can be calculated for all possible values of λ/n for $0 \le \lambda \le 1$. As an example, consider the case for $n=10$ trials. If the observed value of the constant failure rate is $\bar{\lambda} = 0.4$

then $P(\lambda/n)$ values can be calculated and the associated probability distribution illustrated as shown in Figure 6.2. Once this distribution is known, specific confidence levels can be determined. For example, a 95% confidence interval implies that 95% of the failure rate distribution is included within the 95% confidence interval limits. This is shown in Figure 6.2. Similarly, for a 99% confidence level, 99% of the distribution is included within its respective limits. Since constant failure rate values other than $\bar{\lambda} = 0.4$ are of interest, this exercise could be repeated for all constant failure rates of interest.

To use this information more effectively, the probability distributions can be represented graphically in such a way as to illustrate each distribution all at once, thereby producing a more unified picture. An example is shown in Figure 6.3 where the failure rate distributions for $0 \leq \bar{\lambda} \leq 1$ and $n = 10$ trials is plotted. It illustrates both regions where failure rate distribution values are within confidence limits and where failure rate distribution values are outside confidence limits. Since $P(\lambda/n)$ values are not calculated for regions between given (discrete) $\bar{\lambda}$ values, a smooth line is drawn between the calculated $P(\lambda/n)$ values, and is considered to "approximate" the values of interest. (The discrete binomial probability distribution would actually yield a "stair-step" curve when plotted

graphically). Clopper and Pearson generated a standard set of similar charts for two levels of confidence, 95% and 99%, but for various values of n instead of just one [CLOP34]. An example plot for 95% confidence is shown in Figure 6.4. More detailed discussion on this technique, accompanied by some examples is provided in Appendix C.

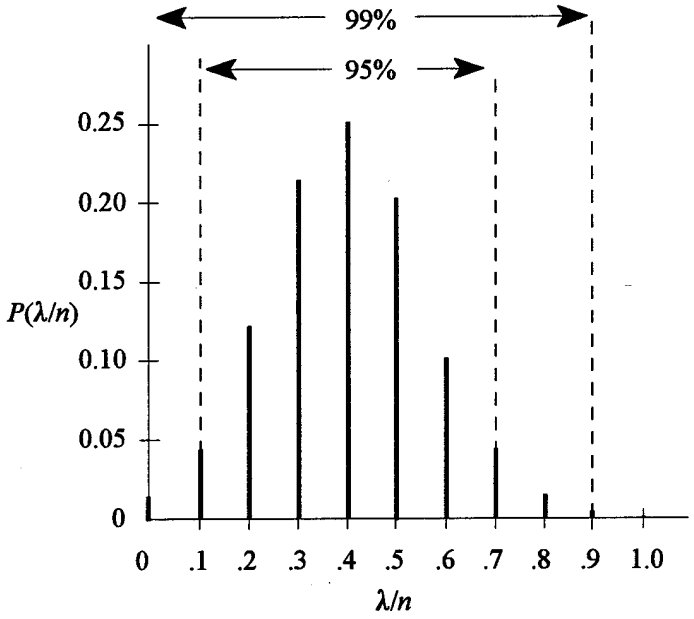


Figure 6.2 - Probability Distribution, $\bar{\lambda} = 0.4$, $n=10$; 95% and 99% Confidence Intervals Marked [WALK53]

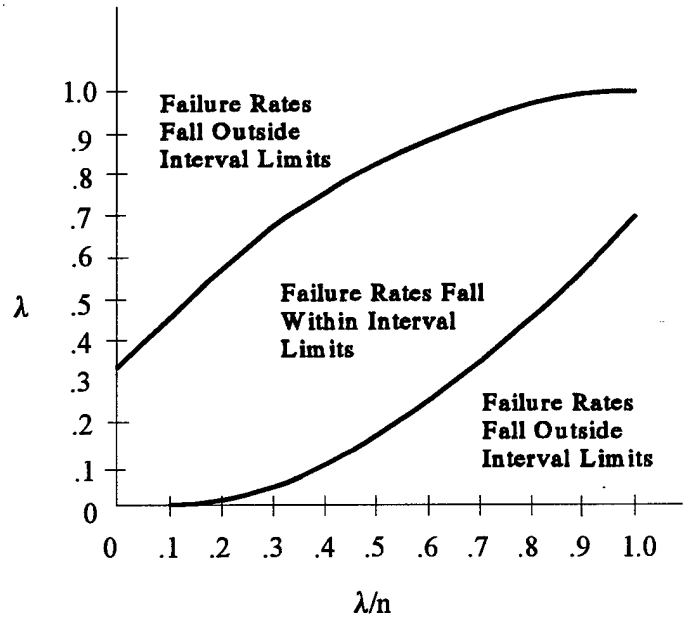
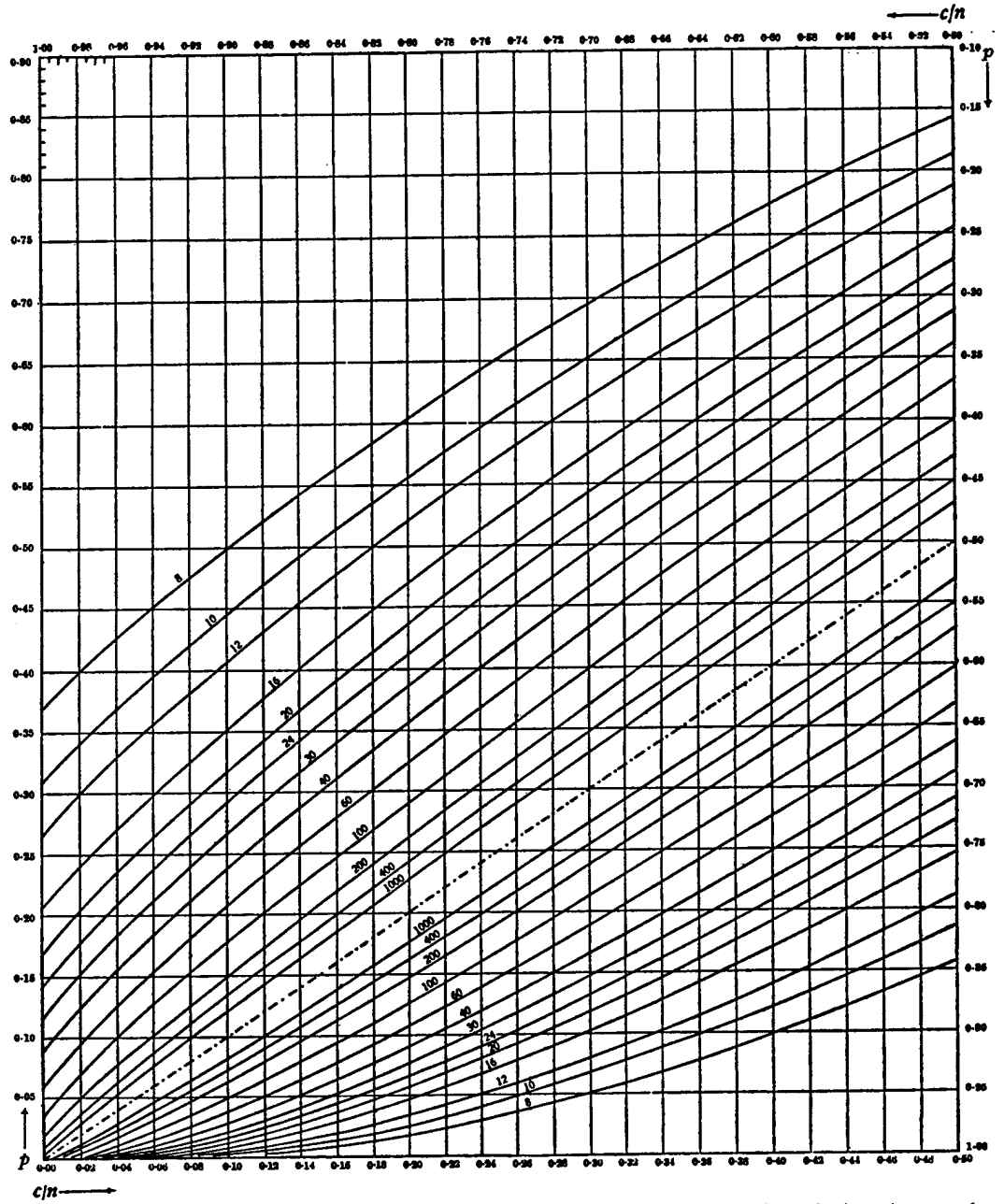


Figure 6.3 - 95% Confidence Interval Chart, $n=10$ [WALK53]



The numbers printed along the curves indicate the sample size n. If for a given value of the abscissa c/n, p_A and p_B are the ordinates read from (or interpolated between) the appropriate lower and upper curves, then

$$\Pr\{p_A \leq p \leq p_B\} \geq 1 - 2\alpha.$$

Figure 6.4 - Confidence Interval Limits for 95% Confidence Level [PEAR76] (©1976 Biometrika, Reprinted with Permission)

6.4 Estimates and Assumptions

To project the reliability of the cache design and the prefetch architecture in reducing $t_{ea\text{-}wc}$, several assumptions are made. Since the actual prefetch architecture is not simulated in its entirety, the reliability of specific cache memory architectures are projected. This is because the design of the cache memory forms the backbone of any prefetch architecture and directly affects its real-time performance. Failure rate estimates and subsequent reliability calculations are performed using IMD output data from the cache simulator for each of the cache architectures listed in Table 6.1. The parameters associated with these architectures result from the cache design guidelines outlined in Chapters 4 and 5.

In addition to the cache designs chosen for success (i.e., no *IMD*=0 occurrences observed), a test measurement is made for a cache architecture known to exhibit behavior that results in "failures" (*IMD*=0 occurrences). Upon examination of all IMD data generated for the various cache designs and program benchmarks, a test case is chosen. For the case of a 32k unified, 2-way associative cache, with a block size of 1024 bytes, both successful and unsuccessful cache behavior occurs when executing the *LRCprl.c* program. As illustrated in the shaded cells of Tables 6.2 and 6.3, depending on the input data (*.ok* or *.fail*), two

Table 6.1 - Cache Parameters Used for Reliability Measurements						
Measure-ment	Cache Type	Benchmark	Cache Size (bytes)	Block Size (bytes)	Associativity	Bus Topology
1*	unified	LRCpr1.c	32k	1024	2-way	single
2	split	SHUTTLE.c	64k	512	2-way	single
3	split	SHUTTLE.c	64k	1024	2-way	single
4	split	LRCpr1.c	64k	512	2-way	single
5	split	LRCpr1.c	64k	1024	2-way	single
6	split	SHUTTLE.c	128k	512	2-way	single
7	split	SHUTTLE.c	128k	1024	2-way	single
8	split	LRCpr1.c	128k	512	2-way	single
9	split	LRCpr1.c	128k	1024	2-way	single
* Test case of known random failures						

independent simulations show $IMD=0$ occurrences of zero (success) and 23 (failure). By randomly generating input data for the program executing on a cache with these design parameters, a number of random successes and failures will result.

Since the goal of the research is to develop a memory system to support hard real-time systems, the estimated reliability of any system should approach 100%, with a corresponding constant failure rate approaching zero ($\bar{\lambda} = 0$). Since no failures are expected (other than for Measurement 1 - the failure test case), a

confidence level is given to each reliability value based on the constant failure rate estimate of $\bar{\lambda} = 0$. To gain the most confidence in the reliability measurements (make the confidence interval as narrow as possible), 1000 program iterations are executed for each reliability measurement using independently generated input files.[20] The corresponding IMD output files of each execution of the program are examined to determine if a failure occurred (any *IMD*=0 occurrences).

Table 6.2 - IMD=0 Count, Cache Size=32k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	112	95	78	78	81	80	80	80	80	80
32	57	45	45	45	45	34	33	33	33	33
64	17	7	7	7	7	7	7	7	7	7
128	2	1	1	1	1	1	1	1	1	1
256	2	0	0	0	0	0	0	0	0	0
512	12	0	0	0	0	0	0	0	0	0
1024	12	0	0	0	0	0	0	0	0	0

[20] This in comparison to the IMD data generated in Section 4.2.3 which is for *one* iteration of the benchmark program using the given cache parameters. To ensure independent Bernoulli trials for failure rate estimates and reliability calculations, input data to the programs are randomly generated.

Table 6.3 - IMD=0 Count, Cache Size=32k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	121	108	89	88	88	91	90	90	90	90
32	48	53	35	35	35	36	35	35	35	35
64	17	8	8	8	8	8	8	8	8	8
128	3	2	2	2	2	2	2	2	2	2
256	3	1	1	1	1	1	1	1	1	1
512	23	2	0	0	0	0	0	0	0	0
1024	23	23	0	0	0	0	0	0	0	0

6.5 Reliability Measurement Results

The reliability measurement results for each of the nine measurements are shown in Table 6.4. Measurement 1 is the case for a known failure. After 1000 program executions, the constant failure rate is 0.854, or $\bar{\lambda}=0.854$. Using this value, an interval can be constructed for 95% and 99% degrees of confidence. Using the confidence interval charts [CLOP34],[21] for 95% confidence, $\lambda_1=0.825$ and $\lambda_2=0.88$; for 99% confidence, $\lambda_1=0.825$ and $\lambda_2=0.89$. Using the equation for reliability, $R(t)=e^{-\lambda t}$, the corresponding reliability confidence intervals are

[21] See Section C.4.1 in Appendix C for a detailed discussion of confidence intervals.

calculated and shown in the last two columns of Table 6.4. Knowing $\bar{\lambda} = 0.854$ in advance of further measurements, one can now say with 95% confidence that the reliability of any one execution of the program is somewhere between 41.4% and 43.8%. Similarity, with 99% confidence, the reliability of the system for one execution of the program is between 41.0% and 43.8%.

For the remaining eight measurements, $\bar{\lambda} = 0$, so the corresponding failure rate values for $n=1000$ and 95% are $\lambda_1=0$ and $\lambda_2=0.003$, and for 99% confidence, $\lambda_1=0$ and $\lambda_2=0.0055$ [CLOP34]. These result in intervals of $0.997 \le R(1) \le 1.0$ and $0.994 \le R(1) \le 1.0$ for 95% and 99% confidence respectively.

6.6 Conclusions

The methods presented in this chapter provide a means for estimating failure rates and projecting the reliability of real-time systems. It gives the designer of such systems a means of quantitatively comparing the architectures of various memory subsystems.

The context of applications envisioned for the use of the theory described in this dissertation are embedded real-time systems with relatively short lifetimes, such as a navigation computer guiding a tactical missile. Since failure rate

Table 6.4 - Reliability Measurement Results			
Measurement	$\bar{\lambda}$	Reliability $R(1)$, Confidence Interval	
		95% CI	99% CI
1*	0.854	$0.414 \leq R(1) \leq 0.438$	$0.410 \leq R(1) \leq 0.438$
2	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
3	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
4	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
5	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
6	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
7	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
8	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
9	0.0	$0.997 \leq R(1) \leq 1.0$	$0.994 \leq R(1) \leq 1.0$
* Test case of known random failures			

estimates and subsequent reliability calculations depend heavily on sample size (number of observed program executions), the use of the reliability techniques discussed lend themselves to applications with shorter lifetimes where an appropriate sample size can be generated and observed. The approach discussed may be not applicable, for example, in analyzing the reliability of applications of very long time periods such as deep space probes, where the number of program executions approaches ∞.

A final note, confidence interval charts such as that illustrated in Figure 6.4, are not intended to provide extremely precise readings, due to the subjective interpretation of specific λ values. The broad picture which they give of the relation between n, $\bar{\lambda}$, λ_1, and λ_2 gives one a good feel for the tradeoffs associated with determining a useful confidence interval by modifying n and $\bar{\lambda}$ and the overall failure rate estimate and reliability of the system under measurement.

Chapter 7 Conclusions

7.1 Introduction

Since their introduction, hierarchical memory systems have been used to improve the overall performance of computing systems by reducing the average latency of the memory subsystem. However, in real-time systems, the "worst case" system performance resulting from the largest, or worst case, memory latency is of primary concern and therefore hierarchical memories are often not used. This research focuses on methods that improve the hierarchical memory subsystem's performance by reducing its worst case effective memory access time, $t_{ea\text{-}wc}$. The memory's $t_{ea\text{-}wc}$ is a function of the cache memory's miss ratio and can also be described in terms of its Cache Reference Inter-Miss Distance (IMD). The IMD is the "distance" between successive cache misses and is related to P_{miss} as $P_{miss} = \frac{1}{IMD + 1}$. From this relationship, the worst case effective memory access time can be also be expressed in terms of IMD as $t_{ea\text{-}wc} = 1 + \frac{T}{IMD_{wc} + 1}$, where T is the time required to access the next (lower) level of the memory hierarchy. The focus of this research is on methods to eliminate the worst case IMD (smallest IMD values) thereby decreasing $t_{ea\text{-}wc}$. If the occurrence of the worst case IMD value can be eliminated, then the worst case program execution time can be reduced in a predictable manner. Methods to control IMD distribution include the

selective choice of cache parameters such as cache type, cache size, block size, and degree of associativity. Simulation of various cache architectures under the same program execution requirements is used to illustrate the ability to modify the cache's IMD distribution by varying cache design parameters. To further aid in eliminating unwanted IMD values, a prefetch architecture is developed. This prefetch architecture employs instruction and data prefetching in addition to cache line/block "freezing" or "locking" to eliminate or "hide" small IMD values. Once specific architectures are selected, failure rate estimates and the subsequent reliability of each design can be projected to determine how well they may function in eliminating small IMD values, and to illustrate the robustness of using techniques developed in this research to support real-time applications.

7.2 Summary of Results

This research is divided into three parts. The first involves modifying specific cache parameters to determine if a relationship exists between the choice of cache parameters and the resulting IMD distribution. These parameters include cache size and type (unified/split), block size, and associativity. After numerous simulations were conducted, it was determined that it *is possible* to eliminate specific IMD values by modifying cache parameters. By increasing both the block and cache size, the count for IMD values of 0, 1, and 2 are significantly reduced.

In addition, an asymptotic limit for both block size and degree of associativity exists in each case for which no additional reduction in IMD can occur regardless of how much either parameter is increased.

The second part of the research involves developing an architecture that facilitates prefetching code and data from main memory in an attempt to anticipate (and therefore "hide") specific cache misses. The design of this architecture is based on findings from the first part of the research in addition to some well-known cache partitioning and protection techniques (see Sections 2.3 and 2.4).

The third and final part of the research investigates the application of reliability theory to estimate the robustness of those techniques used to control the distribution of IMD values. The projected reliability of the memory subsystem of real-time systems employing the cache design guidelines established in Chapters 4 and 5, are shown to be between 99.7% and 100% with a 95% degree of confidence, and between 99.4% and 100% with a 99% degree of confidence.

7.3 Research Contributions

As discussed previously, current design methods for hard real-time computer systems encourage the avoidance of hierarchical memories, and specifically caches. As Hand notes, "If pipelines and caches lead to nonpredictable behavior, stay away from them" [HAND89]. Additionally Simpson recommends,

"One way to address the cache drawback, in applications that require a high degree of determinism, is to run the RISC chip in 'uncached mode' for the sections of application code that demand absolute predictability, or guaranteed response times" [SIMP89]. Finally, Kirk gives an operational example of the Navy's AGEIS Combat System which includes the AN/UYK-43 computer and its 32k-word cache. He says, "... due to unpredictable cache performance, all module utilizations are calculated as if the cache were turned off. As a result, the theoretically overutilized CPU is often underutilized at run-time when the cache is enabled" [KIRK90].

Clearly, the opportunity to impact the area of real-time computer architecture exists. To this point in time, designers generally accepted the unpredictability of caches and designed systems without them. The methods discussed in this dissertation allow designers of real-time systems the opportunity to significantly narrow the gap between the current average execution times (with cache) and predictable real-time execution times (without cache) by selectively choosing appropriate cache design parameters in addition to implementing a prefetch architecture. This may result in the increased use of real-time systems for applications with more stringent timing requirements than are currently possible.

A major contribution of this research results from the approach taken to

improve the cache predictability and memory access time problem in real-time systems by examining and modifying cache behavior in terms of cache reference inter-miss distance (IMD). This contrasts to the more traditional approach of focusing on cache hit or miss ratios. By viewing the problem in terms of IMD, more information is provided on the behavior of the cache than if only the miss ratio is known. Knowledge of P_{miss} only specifies how often cache misses occur, or their *frequency*. IMD however, specifies *where* the cache misses occur with respect to one another in addition to their frequency. Knowing both the *distribution* and the *frequency* of cache misses more accurately describes cache behavior and allows for possible solutions to the real-time cache problem that might not otherwise be possible. Providing more than one view of the problem generally allows more flexibility in developing efficient solutions.

7.4 Future Work

The research presented in this dissertation illustrates that the reduction of worst case effective memory access time and the associated increase in processor performance is possible through the elimination of small IMD values. However since this research is limited in scope, several topics related to this work were not explored. These topics include:

- the effect of external interrupts on IMD distributions and the resulting

impact on real-time interrupt processing speed, efficiency, and scheduling

- use of IMD theory to improve virtual memory system performance

- application of IMD theory to improve real-time system performance using multi/parallel processors with multiple cache memories (cache coherence).

- use of reliability theory to evaluate the presence of hardware and software defects not detected during the initial phase of design (characterized as the "initial failures region" of the hazard rate curve)

Appendices

Appendix A Cache Associativity

A.1 Introduction

In this section, a general discussion of cache associativity is presented. It is worthwhile examining associativity because the choice of associativity implementation greatly impacts the cache's overall effective memory access time.

The degree of associativity of a cache is the number of block frames (or places) in which a given block of information (instructions or data) may reside. Reducing the degree of associativity allow fewer block frames from being searched on a cache reference, thereby reducing the time required for the search. However, this constrains which blocks may be simultaneously resident in the cache [HILL88]. The terms *fully-associative*, *set-associative*, and *direct-mapped* express the restrictions placed on where a block from main memory may reside. A *fully-associative* cache is one in which a block from memory may be placed at any location in the cache. If a block can only be placed in a restricted set of locations, then the cache is called *set-associative*. If a block can only be placed in a single, specific location in the cache, then the cache is called *direct-mapped*. The range of caches from direct-mapped to fully-associative is a continuum of levels of associativity. A direct-mapped cache is a one-way set-associative cache and a fully-associative cache with n blocks is called an n-way set-associative cache

[HENN90]. The area of interest to this research is the relationship between the degree of associativity of the cache and IMD behavior.

A.2 Direct-Mapped Cache (1-Way Set-Associative)

As discussed earlier, a direct-mapped cache requires a block from main memory be placed in only a single, specific location in the cache. Its advantages include simpler implementation and faster access times. Disadvantages include lower hit rates (and a greater number of small IMD values) for smaller sized caches. An example of a unified direct-mapped cache is shown in Figure A.1

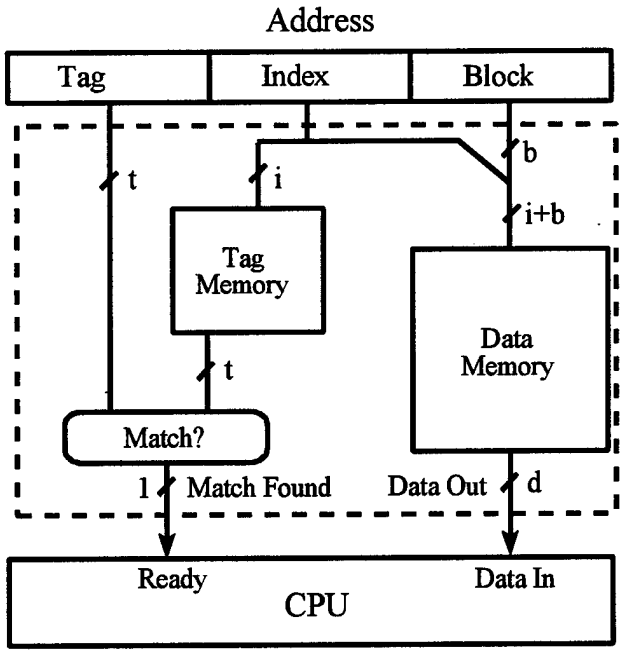


Figure A.1 - Direct-Mapped Cache [HILL88]
(©1988 IEEE, Reprinted with Permission)

(enclosed by the dashed box). The data memory holds all cached instructions and data, while the tag memory hold the address tags associated with a cache block. Each block has a separate tag entry. The tag match logic produces a single bit indicating whether the referenced block is present in the cache and is asserted only if the tag field from the address matches the tag from tag memory.

A direct-mapped cache access requires two parallel actions: 1) accessing the data and passing the word to *Data Out*, and 2) determining if a match has been found (cache hit). The second action requires two steps, and is the critical timing path. The tag memory must first be accessed, and then a comparison of tags must be made. As shown in the next section, a direct-mapped cache access is simpler and faster than a set-associative access because accessing data memory and determining if a match is found can proceed independently. In set-associative caches, the result of the match-found influences the data selected [HILL88].

A.3 Set-Associative Cache

An n-way set associative cache (n=2, 4, 8, 16, etc.) is a commonly used cache organization that allows a block to be mapped to any one of n blocks in a "set" in the cache. This flexibility of block placement generally yields better hit

ratios (especially in smaller caches), but requires checking n blocks during each cache reference. An example set-associative cache is shown in Figure A.2.

To reduce hit time in a set-associative cache, each of the n tags in a set is read and compared to the tags of the reference address in parallel. Each bank in Figure A.2 has the same structure as the direct-mapped cache, but is replicated n times. On a cache reference, the address is passed to all direct-mapped banks. In

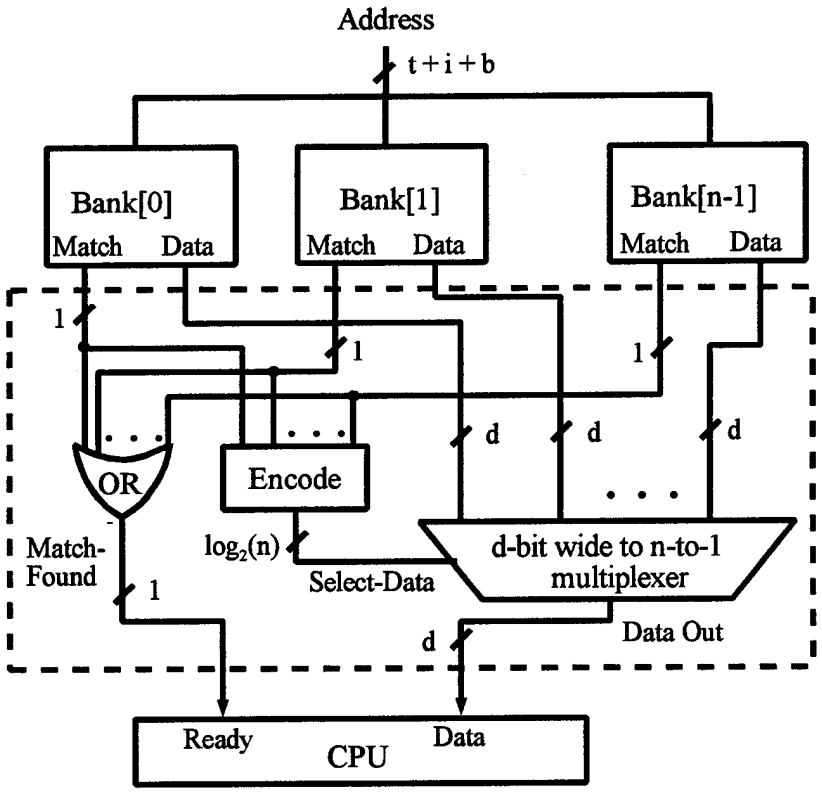


Figure A.2 - Set-Associative Cache (d-bit Word) [HILL88]
(©1988 IEEE, Reprinted with Permission)

parallel, each bank selects a block, sends the data to the multiplexer, and computes whether a match has taken place. After the *n* direct-mapped banks computes the data out and match-found, the set associativity logic (enclosed by the dashed box in Figure A.2) produces a single data out word and match-found signal.

The delay through a set-associative cache is determined by one of three paths:

1) match-found, which signals a cache hit or miss

2) select-data, which selects the data word corresponding to the tag that matched

3) data out, which provides data on a cache hit

A direct-mapped cache has timing paths 1 and 3, but not 2. This is because the location of cache data does not depend on which comparator matched the tag [HILL88]. It is the additional hardware associated with the select-data (enclosed by the dashed line in Figure A.2) that extends the cache access time of a set-associative cache. This timing path will be greater than the critical timing path associated with a direct-mapped cache (assuming identical technology).

Appendix B Prefetching Algorithms and Implementation Methods

B.1 Introduction

Nearly all modern computers use some form of instruction prefetching to improve processor performance. Some also prefetch data as well. Prefetching introduces parallelism into the otherwise sequential operation of von Neumann computers by allowing instruction fetch and execution to proceed concurrently.

The problem with prefetching as a processor accelerator is that program flow is not always predictable. The contents of the queue or cache may have to be discarded when conditional or computed jumps are encountered. This problem can be difficult to avoid since the outcome of a jump may not be known until the instruction immediately proceeding it has been executed. The jump problem carries with it a double performance penalty: first, the processor must wait for the correct next instruction/data to be fetched, and second, the memory access used to fetch the out-of-sequence instructions are lost. The former problem degrades processor performance, while the latter wastes the bandwidth of the memory subsystem [McCR91].

Any prefetching scheme has the goal of reducing the processor stall time by bringing instructions and/or data into the cache before it is needed so it can be

accessed in the future without delay. However, if instructions or data are prefetched too far in advance, the risk or "polluting" the cache exists. This occurs when prefetched lines expel other lines from the cache which are more likely to be referenced in the immediate future. Ideally, a perfect prefetching scheme would totally avoid the memory latency time (thereby ensuring infinite IMDs). Practically, the latency can only be reduced since there are several impediments that prevent a perfect prediction of both the instruction stream (i.e., imperfect branch prediction) and the data stream (i.e., dependent addresses).

Prefetching can be triggered either by a hardware mechanism, a software instruction, or a combination of both. The hardware approach detects accesses with regular patterns and issues prefetches at run time, whereas the software approach relies on the compiler to statically analyze programs and insert prefetch instructions [CHEN95].

B.2 Hardware Controlled Prefetching

Hardware controlled prefetching can be classified into one of two categories - spatial or temporal. Spatial prefetching uses information from the current block to determine what to prefetch, while temporal prefetching uses look-ahead decoding of the instruction stream to determine prefetch actions.

In the spatial scheme, prefetching may occur either on every memory

reference (*Always Prefetch*) or when there is a miss on a cache block. The major mechanism used in the latter case is to record the previous memory address in a history table and generate prefetch requests by calculating a "stride" or "distance" between the current address and the previous address. In the spatial scheme, the opportune time to prefetch is not closely linked to the time of next use, but rather as soon as the current block is accessed.

Temporal mechanisms attempt to have data in the cache "just in time" to be used. The address of the data to be prefetched is based on the values of the speculated operands [CHEN95].

B.3 Software Controlled Prefetching

The central idea behind software controlled prefetching is to exploit the compile-time knowledge about the program. At compile time, *FETCH* instructions are inserted into the instruction stream by the compiler, based on anticipated references and detailed information about the memory system. At program run time, a separate functional unit of the CPU - the fetch unit - interprets these instructions and initiates the appropriate memory read instructions.

Making prefetch decisions at compile time (as opposed to run time, as is common for hardware based prefetching schemes) has the advantage of being able to draw upon detailed information about the program's structure. For example,

if the compiler knows or predicts that a sequence of *LOAD* instructions ($LOAD_1$, ..., $LOAD_i$) will be executed, each $LOAD_i$ instruction can be paired with a *FETCH* instruction to access information required by $LOAD_{i+d}$. The offset by which the compiler anticipates the future requirements is referred to as d - the prefetch distance. Its value will depend on memory latency and the time between individual *LOAD* instructions.

B.3.1 Programmer Controlled Software Data Prefetching

Since data prefetching from main memory can be controlled in software by the programmer, examples of such cache management instructions are worth noting. Two processor instruction sets that support data prefetching include the Motorola/IBM *PowerPC* and the DEC *Alpha* microprocessors.

The PowerPC includes an instruction that allows a programmer to prefetch blocks of data from main memory and load them into the data cache. The *Data Cache Block Touch* instruction brings data into the cache, providing the effective address is contained in the virtual memory system's Translation Look-Aside Buffer (TLB). If there is a miss in the TLB, then this instruction is treated as a *no-op* (no operation executed). The syntax for this instruction is

dcbt rA, rB

where the effective address of the data is the sum of the **A** and **B** registers

[POWE93].

Digital Equipment Corporation provides two instructions for prefetching data from memory in the Alpha microprocessor. *Prefetch Data Hint*, and *Prefetch Data - Modify Hint*, each prefetches 512-byte blocks of data in anticipation of their use. They differ in that the latter instruction provides the additional hint that modifications (stores) to some or all of the data is anticipated [ALPH94]. Their syntax is

FETCH

FETCH_M

B.4 Prefetch Algorithms

Any prefetch algorithm implemented in hardware, software, or both, has three major concerns: 1) when to initiate a prefetch, 2) which lines(s)/block(s) to prefetch, and 3) what replacement status to give the prefetched line(s)/block(s) [SMIT82]. In the following descriptions, a prefetching algorithm is characterized by the knowledge it uses to determine these parameters.

B.4.1 Always Prefetch/Greedy Prefetching

Always Prefetch means that for every memory reference, access to line i implies a prefetch access for line $i+1$ [SMIT82]. An extension to this algorithm is *Greedy Prefetching*. The *Greedy Prefetching* approach begins by allocating an

array of n buffers (cache blocks or lines) to use as a FIFO queue. As long as there are empty buffers in the cache, blocks are prefetched as rapidly as possible to fill them. After a start-up delay to allow prefetching to work ahead, the application begins removing blocks to satisfy demands. When the cache is full, each block read by the application triggers the next fetch to fill the newly freed buffer. The greedy algorithm must allocate and fill enough buffers to satisfy demand during the worst case interval in which the consumption exceeds prefetching [STAE93].

B.4.2 Prefetch on Miss (Demand Prefetch)

Prefetch on miss implies that a reference to block i causes a prefetch to block $i+1$ if and only if the reference block i itself was a miss [SMIT82].

B.4.3 Tagged Prefetching

Tagged Prefetching, proposed by Gindele, associates a single bit called a "tag" with each line or block of the cache [GIND77]. This tag is set to one whenever the line is accessed by the program. It is initially zero, and is reset to zero when the line or block is removed from the cache. Any line brought into the cache by a prefetch operation retains its tag value of zero. When a tag changes from zero to one (i.e., when a line is referenced for the first time after prefetching) a prefetch for the next sequential line is initialized. This idea is similar to prefetching on miss only, except that a miss which did not occur because the line

was prefetched (i.e., had there not been a prefetch, there would have been a miss to this line) also initiates a prefetch [SMIT82].

B.4.4 Rate-Based Prefetching

Greedy Prefetching accommodates variations in the consumption rate by blocking the (faster) prefetching process when the buffer becomes full. If the consumption rate is constant, the producer and consumer processes need not communicate so long as both proceed at the same rate. *Rate-based Prefetching* initiates a fetch at periodic intervals without waiting for "buffer available" events. This approach allows the use of fewer buffer resources, since a fetch can target a full buffer, so long as that buffer is emptied before the fetch data overwrites it. Data loss can occur if consumer and producer rates get out of synch [STAE93].

B.4.5 Scripted Prefetching

An expedient method for synchronizing access to stored data is to determine empirically how long the access takes, and schedule the fetch that far ahead of the demand. For example, if an item requested from memory at time t relative to execution of an instruction is ready l seconds later than it was needed, the schedule can be adjusted to fetch the item at time t-l during the next iteration. Static schedules are called *prefetch scripts* when they anticipate storage latency to meet demands, and *scripted prefetching* is defined as an algorithm that initiates fetches

according to a prefetch script. A static script assumes that the application and its underlying storage system are executing deterministically in real-time without interference from other users.

B.4.6 Dynamically Scripted Prefetching

The most efficient use of system resources occurs when complete knowledge is available of what demands will occur, when they will occur, and exactly how long memory accesses will take. If a sufficient portion of resources can be reserved in advance for an application program, a correct prefetch schedule for that particular configuration of allocated resources can be determined. *Dynamically Scripted Prefetching* refers to an algorithm that performs this resource allocation and then finds and uses a dynamically determined prefetch script. *Dynamically Scripted Prefetching* requires a function for computing anticipated latencies for a given access sequence and a given configuration of reallocated resources [STAE93].

B.5 Simulation Results for No Prefetching vs. Always Prefetch

(see following pages for results)

Table B.1 - IMD=0 Count, 16K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	297	340	103	115	158	133	92	99
32	112	85	46	54	39	40	36	38
64	58	86	20	23	14	14	12	11
128	15	80	11	10	6	6	4	4
256	41	80	4	3	3	3	1	1
512	45	389	3	4	3	36	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.2 - IMD=1 Count, 16K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	375	477	131	143	259	261	126	129
32	116	221	63	70	94	128	61	62
64	54	132	26	31	28	39	24	23
128	65	99	7	10	20	20	5	5
256	37	106	5	5	4	4	3	3
512	34	164	2	13	1	4	1	1
NP: no prefetching; AP: *Always Prefetch*								

Table B.3 - IMD=0 Count, 32K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	197	206	95	100	91	97	90	96
32	73	78	37	40	35	37	34	35
64	53	50	13	15	11	11	10	9
128	9	15	5	7	4	3	3	2
256	5	11	2	1	0	0	0	0
512	6	50	0	0	0	1	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.4 - IMD=1 Count, 32K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	237	299	123	124	121	121	120	120
32	77	146	56	59	57	60	56	56
64	45	117	22	19	23	31	23	21
128	61	94	5	7	16	15	6	6
256	36	101	4	3	3	4	3	3
512	33	97	1	4	1	1	1	1
NP: no prefetching; AP: *Always Prefetch*								

Table B.5 - IMD=0 Count, 64k Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	103	110	91	97	90	96	90	96
32	41	45	36	37	34	36	34	35
64	17	17	11	10	10	9	10	9
128	8	11	3	3	3	2	3	2
256	4	7	0	0	0	0	0	0
512	0	9	0	0	0	0	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.6 - IMD=1 Count, 64k Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	122	127	119	120	120	120	120	120
32	58	62	55	58	56	57	56	57
64	23	26	22	19	23	21	23	21
128	6	8	4	5	6	6	6	6
256	4	4	3	3	3	3	3	3
512	1	2	1	1	1	1	1	1
NP: no prefetching; AP: *Always Prefetch*								

Table B.7 - IMD=0 Count, 128K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	91	96	90	96	90	96	90	96
32	34	35	34	35	34	35	34	35
64	10	9	10	9	10	9	10	9
128	3	2	3	2	3	2	3	2
256	0	0	0	0	0	0	0	0
512	0	0	0	0	0	0	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.8 - IMD=1 Count, 128K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	121	121	119	119	120	120	120	120
32	56	57	55	56	56	57	56	57
64	22	9	22	19	23	21	23	21
128	5	6	5	6	6	6	6	6
256	3	3	3	3	3	3	3	3
512	1	2	1	1	1	1	1	1
NP: no prefetching; AP: *Always Prefetch*								

Table B.9 - IMD=0 Count, 256K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	90	96	90	96	90	96	90	96
32	34	35	35	35	34	35	34	35
64	10	9	10	9	10	9	10	9
128	3	2	3	2	3	2	3	2
256	0	0	0	0	0	0	0	0
512	0	0	0	0	0	0	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.10 - IMD=1 Count, 256K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	119	119	119	119	120	120	120	120
32	55	56	55	56	56	57	56	57
64	22	19	22	19	23	21	23	21
128	5	6	5	6	6	6	6	6
256	3	3	3	3	3	3	3	3
512	1	1	1	1	1	1	1	1
NP: no prefetching; AP: *Always Prefetch*								

Table B.11 - IMD=0 Count, 512K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	90	96	90	96	90	96	90	96
32	34	35	34	35	34	35	34	35
64	10	9	10	9	10	9	10	9
128	3	2	3	2	3	2	3	2
256	0	0	0	0	0	0	0	0
512	0	0	1	0	0	0	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.12 - IMD=1 Count, 512K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	119	119	119	119	120	120	120	120
32	55	56	55	56	56	57	56	57
64	22	19	22	19	23	21	23	21
128	5	6	5	6	5	6	6	6
256	3	3	3	3	3	3	3	3
512	1	1	1	1	1	1	1	1
NP: no prefetching; AP: *Always Prefetch*								

Table B.13 - IMD=0 Count, 1024K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	90	96	90	96	90	96	90	96
32	34	35	34	35	34	35	34	35
64	10	9	10	9	10	9	10	9
128	3	2	3	2	3	2	3	2
256	0	0	0	0	0	3	0	0
512	0	0	0	0	0	0	0	0
NP: no prefetching; AP: *Always Prefetch*								

Table B.14 - IMD=1 Count, 1024K Unified/Split Caches, Benchmark: SHUTTLE.c								
Block Size (bytes)	Associativity							
	Unified Cache				Split Cache			
	1-way		2-way		1-way		2-way	
	NP	AP	NP	AP	NP	AP	NP	AP
16	119	119	119	119	120	120	120	120
32	55	56	55	56	56	57	56	57
64	22	19	22	19	23	21	23	21
128	5	6	5	6	6	6	6	6
256	3	3	3	3	3	3	3	3
512	1	1	1	1	1	1	1	1
NP: no prefetching; AP: *Always Prefetch*								

Appendix C Reliability

C.1 Reliability Function

The probability of failure as a function of time, t, can be defined as

$$P_r(T \le t) = \int_0^t f(\upsilon)\, d\upsilon = F(t), \qquad t \ge 0 \tag{54}$$

where T is a random variable that denotes failure time, t is the time over which the probability is measured, and $f(\upsilon)$ is the probability density function of T. In general, $f(\upsilon)$ will change as environmental conditions change. $F(t)$ is the probability that the device will fail by time t. Time can be replaced by other measures of interest such as cycles, stress, or (in the case of immediate interest) number of independent program executions. Since reliability is defined as the probability of success, the reliability function, $R(t)$, can be written as

$$R(t) = P_r(T \ge t) = \int_t^\infty f(\upsilon)\, d\upsilon = 1 - F(t) \tag{55}$$

The Mean Time to Failure (*MTTF*) is the expected time during which the system or component will perform successfully [MART82], and is expressed as

$$MTTF(T) = \int_0^\infty R(t)\, dt \tag{56}$$

For this research, MTTF is expressed in terms of the number of program runs

before failure or Mean Program Runs Between Failure (MPBF).

C.2 The Binomial Distribution

A binomial distribution can be used to describe the success or failure of a system for a given sample size (number of trails). The binomial distribution is a discrete probability distribution associated with events that have a total of two possible outcomes. Much early research done on probability theory characterized by the binomial distribution was done by several generations of the Bernoulli family. As a result, the Bernoulli name has come to be associated with this class of experiment, and each repetition of an experiment involving only two outcomes is called a "Bernoulli trial." For the purposes of probability theory used for the research described in this dissertation, interest centers not a single Bernoulli trial, but several independent, repeated Bernoulli trials. The "independence" of each trial implies that the outcome of any one trial cannot influence the results of any other trial. In addition, when a Bernoulli trial is "repeated," the conditions under which each trial is held must be an exact replication of the conditions underlying all other trials. The "independence" and "repeatability" associated with experiments conducted during this research are discussed later.

The binomial distribution is described by the values of n and p, which are referred to as the parameters of the distribution. Each parameter is a characteristic

of a population, where n is the "number of trials" and p is the "probability of success (or failure) on a single trial." Given specific values of n and p, the probability of any specified number of successes (or failures), $P(x/n)$, can be calculated. The value of p is a constant for the population under examination,[22] and is determined by observing or estimating the number of failures prior to calculating $P(x/n)$. The binomial distribution can be described by the following formula:

$$P(x \text{ successes in } n \text{ trials}) = \begin{cases} \binom{n}{x} p^x q^{n-x} & \text{for } \begin{cases} x = 0,1,2,\ldots,n \\ n = 1,2,\ldots \end{cases} \\ 0 & \text{otherwise} \end{cases} \quad (57)$$

where n = number of independent trials

p = probability of success in a single trial

$q = 1 - p$

To illustrate a situation in which the binomial distribution applies, suppose a production process is producing solid-state components that are classified as either "good" or "defective." When the process is not working correctly, there is a constant probability, $p=0.10$, that a component will be defective. In this

---

[22] Consistent with the constant failure rate region of the hazard rate curve in Figure 6.1

situation, the number of defective components, x, can range anywhere from zero up to the total number of components examined, n. The binomial distribution can be used to determine the probability of failure for any specified value of x and n. For example, the probability that exactly one component will be defective ($x=1$) out of a sample of four ($n=4$), and $p=0.10$ is calculated as follows

$$\begin{aligned} P\ (x=1,\ n=4) &= \binom{n}{x} p^x q^{n-x} \\ &= \binom{4}{1} (0.10)^1 (0.90)^3 \\ &= 0.2916 \end{aligned} \tag{58}$$

Similarity, the probability that there are exactly two defective components ($x=2$) out of a sample of $n=4$ can be calculated to be

$$P(x=2,\ n=4) = \binom{4}{2} (0.10)^2 (0.90)^2 = 0.0486 \tag{59}$$

The probability of any number of defective components from zero to four may be determined in the same way [HARN82].

C.3 The Exponential Distribution

The exponential distribution is widely used in describing the reliability of many systems. It can be shown that the exponential distribution accurately

describes the failure time distribution of the *chance* failure region (see Figure 6.1) and is uniquely associated with a constant failure rate [MART82]. The reliability function (55) then becomes

$$R(t) = e^{-\lambda t} \tag{60}$$

where t is the time over which reliability is measured and λ is the constant failure rate. In the specific case examined in this dissertation, λ can be obtained by observing the number of failures over a specified number of program executions. This value of λ can then be used for future reliability calculations. The number of program executions over which the reliability is measured is substituted for t. λ is related to *MTTF* as $\lambda = 1/MTTF$ or $\lambda = 1/MTBF$. For a perfectly reliable system, $R = 1$, so λ must equal zero. Proving λ equals (or approaches) 0 can be a very difficult task, so a more practical (and measurable) approach is taken. For example, if the reliability were lowered to 99% over 100 program runs, solving for λ leads to

$$\lambda = \frac{-\ln R}{t} = 0.1 \times 10^{-3} \tag{61}$$

Therefore, to demonstrate that a particular solution is 99% reliable for 100 independent program runs, a λ value of 0.1×10^{-3} must be obtained [BLAK79].

This implies that no more than 1 failure per 10,000 program runs during testing. Such a reliability value can be obtained by simulating the proposed system, executing the program 10,000 times (using independent input data to ensure Bernoulli trials), and measuring any observed failures. Using these failures as data points, constant failure rates estimates and reliability values can then be projected. However, running a program 10,000 times and collecting data on each execution may prove to be an impractical task in many cases. An alternative method is to calculate reliability with certain "confidence level" using a lesser number of program executions to estimate the constant failure rate, λ. Using observed or estimated failures for a smaller number of program executions as data points, a value of λ can be obtained and indicated as $\bar{\lambda}$. This value for $\bar{\lambda}$ then becomes the constant failure rate used for future probability (and reliability) calculations and is substituted for p in the binomial formula (57). The use of confidence levels in association with constant failure rate estimates will be discussed in the next section.

C.4 Confidence Levels and Intervals

In projecting the reliability of real-time cache architectures, $\bar{\lambda}$ values are obtained by observing failures of program execution time to meet hard real-time deadlines for a number of program executions. For example, if 1,000 program

runs are executed and four failures are observed, $\bar{\lambda} = 4 \times 10^{-3}$. In the case where no failures are observed, $\bar{\lambda} = 0$. In either case, the resulting failure rate estimate may have a "confidence level" associated with it to improve its usefulness. The confidence level is a numerical value, usually between 0.90 and 1.00, that is associated with the constant failure rate, and therefore the reliability. If $\bar{\lambda}$ is based on a random sample of n program executions (trials), then it can be said that the true value of λ will fall within the range of confidence limits a and b, $a \leq \lambda \leq b$ with a certain level of confidence.[23] This range between a and b is described as the "confidence interval." For example, the meaning of a 95% confidence level is that the true parameter (or outcome) is expected to be included within a specific range of values 95 out of 100 times. The confidence interval limits can be considered the best and worst case probabilities, and are based on the best and worst case failure rate estimates and the number of trials (program executions) over which $\bar{\lambda}$ is observed. Once these confidence limits are known, reliability values can then be calculated at each limit, giving a range of possible reliability values.

---

[23] The underlying premise that allows a confidence level to be determined is based on the statistical experience that the more often p lies in the interval $p_1 \leq p \leq p_2$, the higher confidence one can have that it will also lie within that interval in the future.

In general, confidence intervals and their limits are constructed on the basis of sample information, so changes in $\bar{\lambda}$ and n affect the size of the interval and the interval limits. The underlying premise that allows a confidence level to be associated with failure rates (and reliability values) is based on the statistical experience that the more often λ lies in the interval $\lambda_1 \leq \lambda \leq \lambda_2$, the higher confidence one can have that it will also lie within that interval in the future. In addition, the more Bernoulli trial observations are collected, the more confident one can be about the estimate and, as a result, the smaller the interval needed to assure a given level of confidence [HARN82].

The difference between probability and confidence is that the concept of probability is used in reasoning from a known population (number of program executions) to a random sample (number of successes or failures) whereas the concept of confidence is used in reasoning from an observed sample (number of failures over a number of program executions) to its unknown population (failure rate) [WALK53].

C.4.1 Construction of Confidence Intervals

Clopper and Pearson present methods that allow confidence levels to be assigned to binomial probabilities for a range of x and n values [CLOP34]. For

example, a constant probability of success (or failure) p can be observed (or estimated), and can be said to fall into the interval $p_1 \le p \le p_2$ with a certain degree of confidence. The confidence interval limits, p_1 and p_2, are determined by calculating the region defined by p_1 and p_2 that include 95% (for a 95% confidence interval) of all possible outcomes of $P(x/n)$. The resulting confidence interval is a range of binomial probabilities that is calculated for the constant probability of success (or failure) p, and the number of independent trials over which p is observed, n. Using the binomial equation (57), the probability of future successes, $P(x/n)$, can be calculated using p_1 and p_2, resulting in an interval for P, $P_1 \le P \le P_2$. Clopper and Pearson constructed such confidence intervals for two cases: 95% and 99% confidence [CLOP34], [PEAR74]. The basic theory behind their construction is described in the following paragraphs.

Using the binomial equation (57), the probability of x successes in n trials can be calculated for all possible values of x/n for $0 \le x/n \le 1$. As an example, consider the case for $n=10$ trials. If the observed (or estimated) value of the constant probability is 0.4 ($p=0.4$) then $P(x/n)$ values can be calculated and are shown in Table C.1. Since these values cover all possible probabilities for $0 \le x/n \le 1$, their sum will equal 1.0 (1.0001 with rounding errors). This probability distribution is illustrated in Figure C.1 [WALK53]. Given this information, it can

now be determined how confident one is, based on the given constant probability of success, that the resulting calculated probability will fall within a specific range of values. For example, if one wants to work with a region that will give 95% confidence (and 5% failure) that the calculated probability will fall within this region of values (based on n and p), confidence limits must be determined. The choice of these limits implies that 95% of the calculated probability values should fall within the interval defined by these limits. As shown in Figure C.1, the most likely opportunity for success occurs for those $P(x/n)$ values with the greatest magnitude in the distribution. The most likely opportunities for failure will occur at either extreme of the distribution ($x/n \rightarrow 0$ and $x/n \rightarrow 1$). Assuming a 95% confidence interval, 2.5% of the failures will occur at either tail of the distribution. Examining the data in Table C.1 and Figure C.1, the values of x/n at either end of the distribution that cause $P(x/n)$ to be less than 0.025 (2.5%) should be included in the failure region while all others will be in the region of success. Since 5% of the calculated probabilities should fall in the failure region, the sum of individual probability values is made at either tail of the distribution until the 0.025 threshold is reached or exceeded. For $x/n=0$, $P(0/10) = 0.006$ which is less than 0.025. Therefore this value is included in the failure region. If $P(0/10)$ and $P(1/10)$ are examined, their sum is 0.046 which is greater than 0.025. As a

result, $x/n=1$ is not in the failure region, but is included in the region of success.

Table C.1 - Calculated Probabilities for $n=10$ and $p=0.4$

x	$P(x/n)$
0	0.006
1	0.040
2	0.121
3	0.215
4	0.251
5	0.201
6	0.111
7	0.042
8	0.011
9	0.002
10	0.0001

Using the same methodology, the sum of $P(x/n)$ values for $x/n=$ 1.0, 0.9, and 0.8 is 0.0131 which is less than 0.025 and should therefore be included in the failure region [WALK53]. It can now be said that in 95 out of 100 cases, the calculated value for $P(x/n)$ will fall in a region defined by $P(1/10)$ and $P(7/10)$, or $P(1/10)<P(x/n)<P(7/10)$. Likewise, if the confidence interval is expanded to 99%, then 1% of the $P(x/n)$ values are rejected (0.005 at either tail of the distribution). For this example, the interval now becomes

$P(0/10) < P(x/n) < P(9/10)$. Both cases are illustrated in Figure C.2.

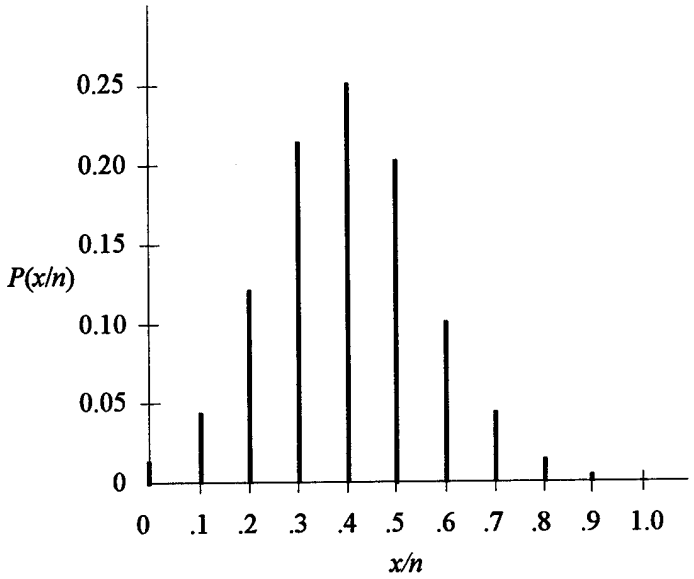


Figure C.1 - Probability Distribution for $P(x/n)$, $n=10$, $p=0.4$ [WALK53] (©1953 Harcourt Brace & Company, Reprinted with Permission)

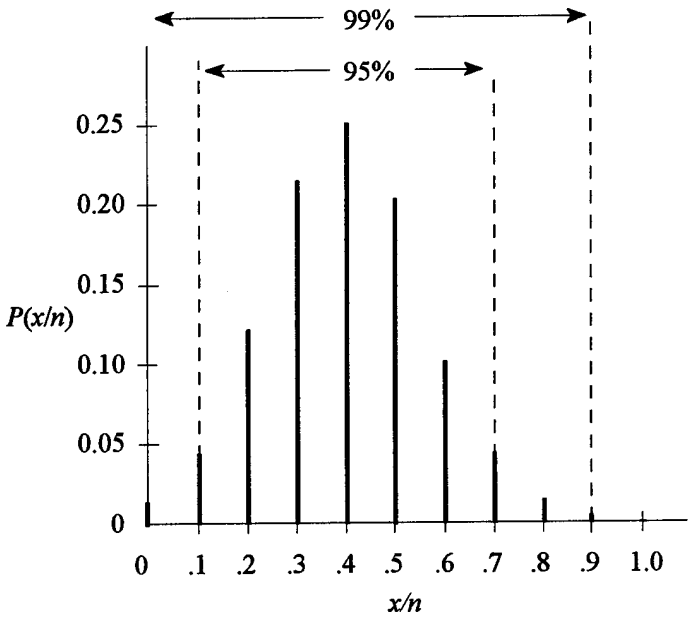


Figure C.2 - Confidence Intervals Marked on Probability Distribution, p=0.4, n=10 [WALK53] (©1953 Harcourt Brace & Company, Reprinted with Permission)

Since constant probability values other than p=0.4 are of interest, this exercise could be repeated for all p values of interest. If all p values were examined and corresponding $P(x/n)$ values calculated, a table of computed probability distributions can be created as shown in Table C.2. While p is constant for any one population, all possible populations are considered, so the scale for p is a continuum extending from 0 to 1. Each horizontal row of the table is a sampling distribution for which the sum is equal to 1.0 (except for rounding errors)

[WALK53].[24]

Table C.2 - Calculated Probability Distributions, $n=10$ [WALK53]

	x/n										
p	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.9						.002	.011	.057	.194	.387	.349
0.8				.001	.006	.026	.088	.201	.302	.268	.107
0.7			.001	.009	.037	.103	.200	.267	.233	.121	.028
0.6		.002	.011	.042	.111	.201	.251	.215	.121	.040	.006
0.5	.001	.010	.044	.117	.205	.246	.205	.117	.044	.010	.001
0.4	.006	.040	.121	.215	.251	.201	.111	.042	.011	.002	
0.3	.028	.121	.233	.267	.200	.103	.037	.009	.001		
0.2	.107	.268	.302	.201	.088	.026	.006	.001			
0.1	.349	.387	.194	.057	.011	.002					

p - constant probability; *x* - number of successes (or failures); *n* - number of trials

To use this information more effectively, the distributions can be represented graphically in such a way as to illustrate each distribution all at once, thereby producing a more unified picture. Figure C.3 shows the distributions of Table C.2 for $0<p<1$ and $n=10$ trials. It illustrates both regions of success and failure for 95% confidence. Since $P(x/n)$ values are not calculated for regions between given (discrete) p values, a smooth line is drawn between the calculated

[24] The vertical columns are *not* probability distributions.

$P(x/n)$ values, and is considered to "approximate" the values of interest. [25] Clopper and Pearson expanded this idea, generating a standard set of similar charts for two levels of confidence, 0.95 and 0.99, but for various values of n instead of just one [CLOP34], [PEAR76]. Example charts for 95% and 99% confidence are reproduced in Figures C.4 and C.5 respectively. These charts can be used to determine confidence interval limits p_1 and p_2 based on the constant probability of failure, p $(\bar{\lambda})$, and the number of trials, n, over which they were observed.[26] Once p_1 and p_2 are determined, the confidence interval for $P(x/n)$ can then be calculated. The failure rates λ_1 and λ_2 can be substituted for p_1 and p_2, thereby creating a failure rate interval. These limits can then be used to calculate a range of reliability values.

---

[25] The discrete binomial probability distribution would actually yield a "stair-step" curve when plotted graphically.

[26] Confidence interval charts are *not* intended to provide extremely precise readings due to the subjective interpretation of specific λ values. The broad picture which they give of the relation between n, $\bar{\lambda}$, λ_1, and λ_2 gives one a good feel for the tradeoffs associated with determining a useful confidence interval by modifying n and $\bar{\lambda}$ and the overall reliability of the system under measurement.

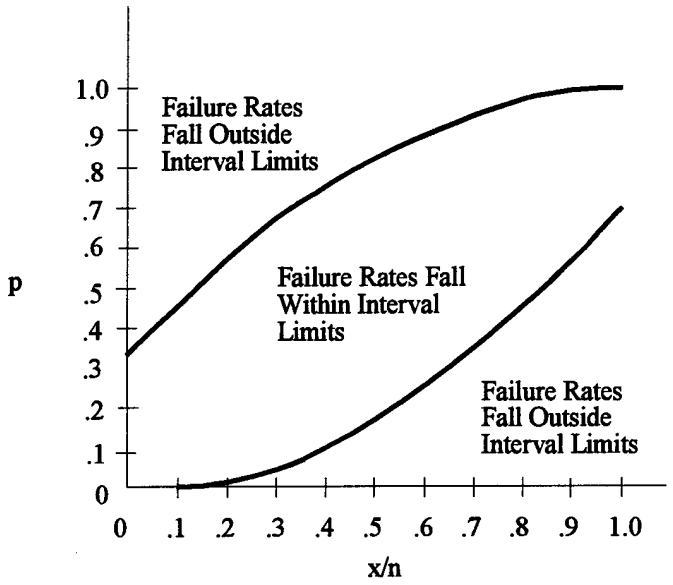


Figure C.3 - 95% Confidence Interval Chart, $n=10$ [WALK53] (©1953 Harcourt Brace & Company, Reprinted with Permission)

C.5 Example Reliability Calculations

The following examples illustrate the use of interval estimation and reliability calculations.

Case 1 - Failures Observed

A program using a randomly generated input file (to ensure Bernoulli trials) is run 100 times ($n=100$ trials), and 2 real-time deadline failures are observed resulting in an MPBF of 50 program executions between failures. The resulting constant probability of failure is estimated to be 0.02 or 2%, with $\bar{\lambda}=0.02$. With

what confidence can this failure rate estimate be used? Using the confidence interval chart for 95% confidence shown in Figure C.4, it may be determined that this failure rate value falls into the interval of $\lambda_1 \leq 0.02 \leq \lambda_2$, where λ_1=0.0 and λ_2=0.07. This says that "with a 95% degree of confidence, the true value of the constant failure rate of the system falls between 0 and 0.07." Using these failure rate limits of λ_1 and λ_2, a reliability interval can be calculated as $0.932 \leq R(1) \leq 1.0$. Similarly as stated above, it can be said that "with a 95% degree of confidence, the true value of the system's reliability for one execution of the program falls between 93.2% and 100%." Additional reliability intervals are calculated by varying the number of program executions over which *R(t)* is measured while keeping $\bar{\lambda} = 0.02$ and n=100 constant [CLOP34], [PEAR76]. The results are shown in Table C.3.

If λ is estimated by observing the number of failures over a larger sample size of program executions, then the corresponding confidence interval will shrink, giving a better estimate of the overall system reliability. For example, if the number of failures observed over 1000 program runs is 20, then $\bar{\lambda} = 0.02$ as before, but it can be said with more confidence, since a larger sample (n=1000 vs. n=100) is used. Using Figure C.4 (95% confidence), λ_1=0.0125 and λ_2=0.0275

results in a reliability interval of $0.973 \leq R(1) \leq 0.988$. This is a much smaller reliability interval, giving a more useful value for overall reliability. Corresponding reliability intervals for various values of t (number of executions) are shown in Table C.3. If n were taken to the extreme ($n=\infty$), and $\bar{\lambda} = \lambda_1 = \lambda_2 = 0.02$, the reliability interval would be zero and a specific reliability value for each case would be available (instead of a range of values) . They are shown in the last column of Table C.3. Note how increasing the degree of confidence results in a wider interval for all cases. Table C.4 shows the values obtained for a 99% confidence level and $\bar{\lambda} = 0.02$.

Case 2 - No Failures Observed

For this case, no failures are observed for $n=100$ trails. This results in an estimate of $\bar{\lambda} = 0$. The resulting reliability is then

$$R(t) = e^{-0t} = 1.0 \qquad (62)$$

for all t (any number of program executions). Using $\bar{\lambda} = 0$, $n=100$ program runs, and a 95% confidence level (Figure C.4), two confidence interval limits are found: $\lambda_1=0$, and $\lambda_2=0.03$. Corresponding λ values for a 99% confidence level are $\lambda_1=0$, and $\lambda_2=0.055$. If n is increased to 1,000 executions, corresponding λ

limits for 95% confidence are $\lambda_1=0$ and $\lambda_2=0.003$, while for 99% confidence $\lambda_1=0$ and $\lambda_2=0.0055$. The resulting reliability intervals for 95% and 99% and $\bar{\lambda}=0$ are shown in Tables C.5 and C.6 respectively.

Table C.3 - Reliability Intervals for 95% Confidence Level, $\bar{\lambda}=0.02$

	n=100		n=1,000		n=∞
Runs, t	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)$
1	0.932	1.0	0.972	0.987	0.980
10	0.496	1.0	0.759	0.882	0.818
50	0.030	1.0	0.252	0.535	0.367
100	0.001	1.0	0.063	0.286	0.135
200	8.31×10^{-7}	1.0	0.004	0.082	0.018
500	6.30×10^{-16}	1.0	1.06×10^{-6}	0.002	4.52×10^{-5}
1000	3.97×10^{-31}	1.0	1.14×10^{-12}	3.76×10^{-6}	2.06×10^{-9}

Table C.4 - Reliability Intervals for 99% Confidence Level, $\bar{\lambda} = 0.02$					
	$n=100$		$n=1{,}000$		$n=\infty$
Runs, t	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)$
1	0.916	0.998	0.968	0.990	0.980
10	0.416	0.987	0.722	0.904	0.818
50	0.012	0.939	0.196	0.606	0.367
100	1.58×10^{-4}	0.882	0.038	0.367	0.135
200	2.51×10^{-8}	0.778	0.001	0.135	0.018
500	9.99×10^{-20}	0.535	8.76×10^{-8}	0.006	4.53×10^{-5}
1000	9.98×10^{-39}	0.286	7.68×10^{-15}	4.53×10^{-5}	2.06×10^{-9}

Table C.5 - Reliability Intervals for 95% Confidence Level, $\bar{\lambda} = 0$					
	$n=100$		$n=1{,}000$		$n=\infty$
Runs, t	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)$
1	0.970	1.0	0.997	1.0	1.0
10	0.740	1.0	0.970	1.0	1.0
50	0.223	1.0	0.860	1.0	1.0
100	0.049	1.0	0.740	1.0	1.0
200	2.47×10^{-4}	1.0	0.548	1.0	1.0
500	3.05×10^{-7}	1.0	0.223	1.0	1.0
1000	9.35×10^{-14}	1.0	0.049	1.0	1.0

Table C.6 - Reliability Intervals for 99% Confidence Level, $\bar{\lambda}=0$					
	$n=100$		$n=1{,}000$		$n=\infty$
Runs, t	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)_{Lower}$	$R(t)_{Upper}$	$R(t)$
1	0.946	1.0	0.994	1.0	1.0
10	0.576	1.0	0.946	1.0	1.0
50	0.063	1.0	0.759	1.0	1.0
100	0.004	1.0	0.576	1.0	1.0
200	$1.67\text{x}10^{-5}$	1.0	0.332	1.0	1.0
500	$1.13\text{x}10^{-12}$	1.0	0.063	1.0	1.0
1000	$1.29\text{x}10^{-24}$	1.0	0.004	1.0	1.0

As can be seen from these two examples, the sample size n (number of statistically independent program executions over which $\bar{\lambda}$ is obtained), is crucial in obtaining a good estimate of the system reliability. For example, when measuring the reliability over 200 program executions and using $\bar{\lambda}=0.02$ over $n=100$, $n=1{,}000$, and $n=\infty$, different intervals are obtained. For example, using 95% confidence leads to the following:

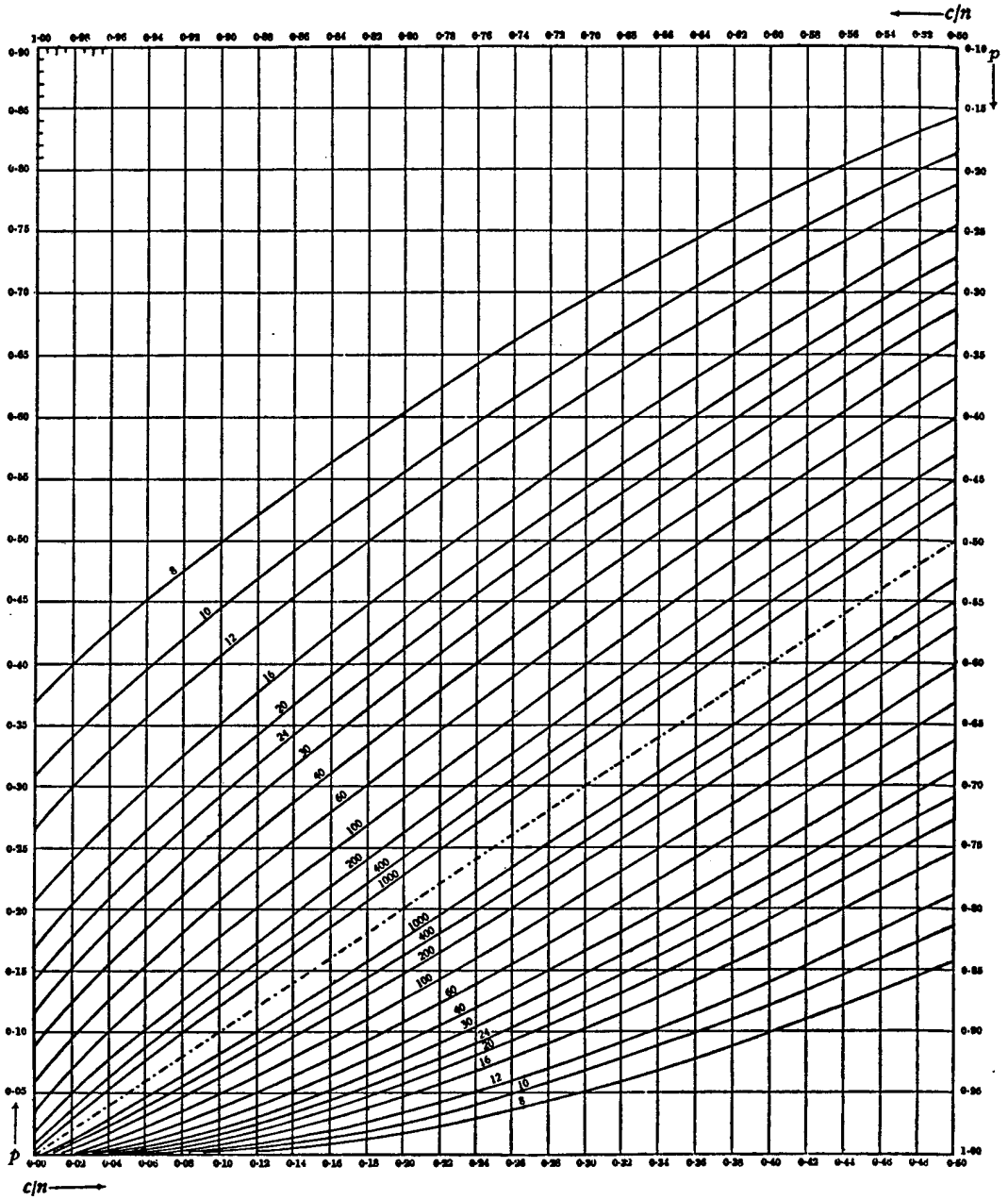
$$\begin{aligned} 8.31x10^{-7} \le R(200) \le 1.0 \quad & \text{for } \bar{\lambda} = 0.02,\ n = 100 \\ 0.004 \le R(200) \le 0.082 \quad & \text{for } \bar{\lambda} = 0.02,\ n = 1{,}000 \\ R(200) = 0.018 \quad & \text{for } \bar{\lambda} = 0.02,\ n = \infty \end{aligned} \tag{63}$$

By increasing the sample size by an order of magnitude over which $\bar{\lambda}$ is obtained, ($n=100$ to $n=1000$), the resulting reliability confidence interval decreases by almost 13 times. The maximum % error (difference between R(t) for $\bar{\lambda}$ measured at $n=\infty$ and $\bar{\lambda}$ measured at $n=100$ or $n=1000$) can be calculated by examining respective R(200) values at $\bar{\lambda}_1$ and $\bar{\lambda}_2$,

$$\text{maximum \% error} = \max \{(R(t)_{\lambda_2} - R(t)_{\infty}),\ (R(t)_{\infty} - R(t)_{\lambda_1})\} \times 100\% \quad (64)$$

With 95% confidence and for $n=100$, the maximum error = 98.2%; for $n=1000$, the maximum error = 6.4%; and for $n=\infty$, the maximum error=0%.

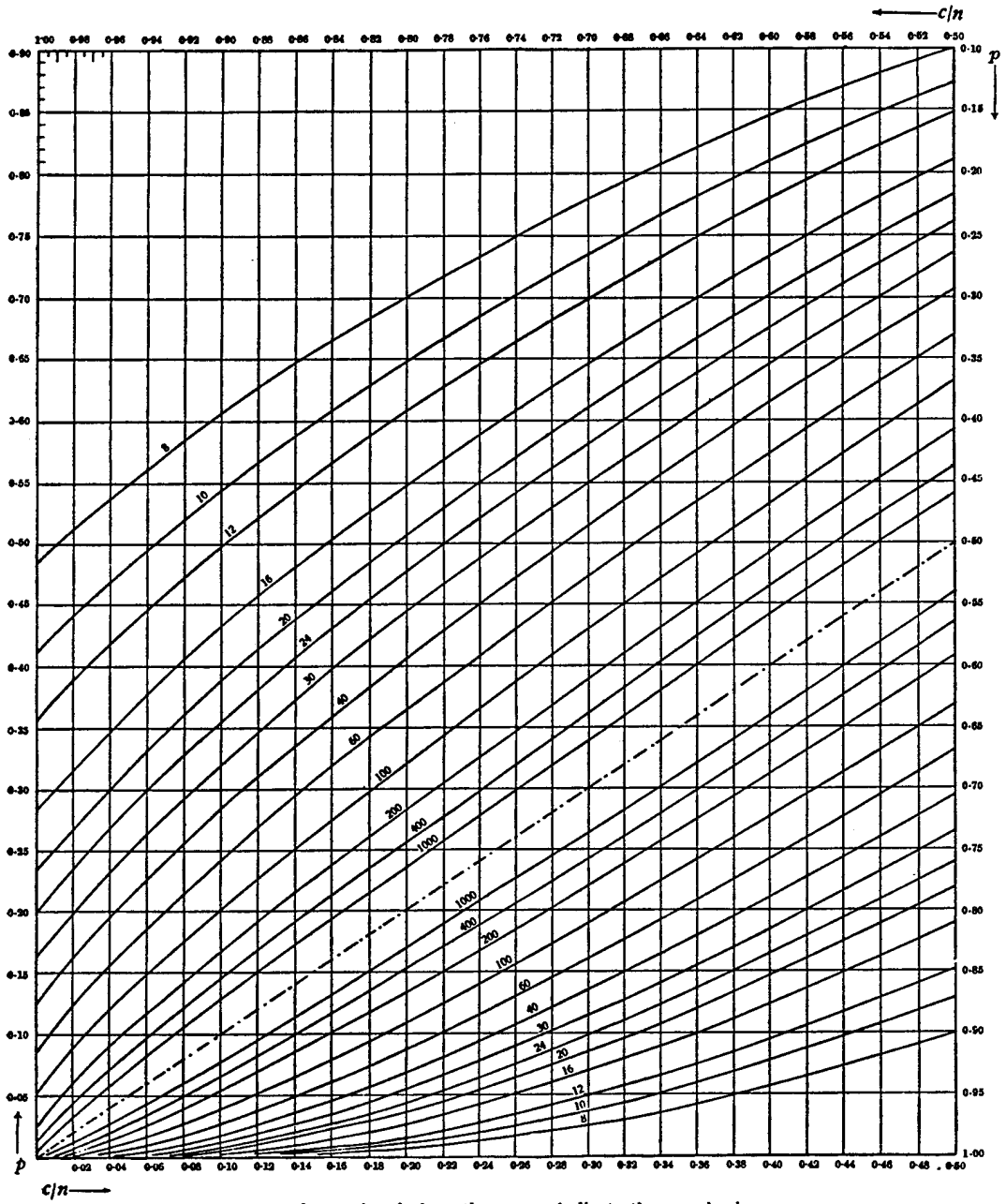
Attempting to observe $n=\infty$ samples to determine $\bar{\lambda}$ is impossible, so something less is required. For $n=1{,}000$ a maximum error of 6.4% results, and would seem acceptable for most cases when compared to the 98.2% error for $n=100$. Increasing n by an additional order of magnitude to $n=10{,}000$ may gain a more accurate reliability value, but may not be worth the additional resources and time required since it would only decrease the maximum error by an amount less than 6.4%.



The numbers printed along the curves indicate the sample size n. If for a given value of the abscissa c/n, p_A and p_B are the ordinates read from (or interpolated between) the appropriate lower and upper curves, then

$$\Pr\{p_A \leq p \leq p_B\} \geq 1 - 2\alpha.$$

Figure C.4 - Confidence Interval Limits for 95% Confidence Level [PEAR76] (©1976 Biometrika, Reprinted with Permission of Biometrika Trustees)



The numbers printed along the curves indicate the sample size n.

Note: the process of reading from the curves can be simplified with the help of the right-angled corner of a loose sheet of paper or thin card, along the edges of which are marked off the scales shown in the top left-hand corner of each Chart.

Figure C.5 - Confidence Interval Limits for 99% Confidence Level [PEAR76] (©1976 Biometrika, Reprinted with Permission of Biometrika Trustees)

Appendix D Performance Evaluation Tools

D.1 Evaluation Tools

Processor and cache performance evaluation tools can be roughly divided into two types of models: *trace-driven* models and *execution-based* models. Trace-driven models simulate the instruction and data flow of the processor and cache, but in the case of a processor model, do not actually execute instructions or generate results. Instead, they are fed a program trace which contains the dynamic sequence of instruction addresses, instruction opcodes, and data addresses that occur during execution. Execution-based models, on the other hand, execute code much as a real processor would, generating results and using those results in conditional branches and so forth [POUR94]. For this work, trace-driven cache simulation was used.

Trace-driven simulation uses one or more (instruction and data address) traces and a cache simulator. A trace is a log of a dynamic series of memory references, recorded during the execution of a program or workload. The information recorded includes the address of the reference and the reference's type (instruction fetch, data read, or data write). One or more traces are then used to drive a simulation model of cache memory. A cache simulator is a program that accepts a trace and parameters that describe a unified or split cache, mimics the

behavior of the cache in response to the trace, and computes performance metrics (i.e., miss ratio) [HILL89]. It also can provide information on a cycle by cycle basis of cache activity such as hit or miss and memory address. By varying parameters of the cache simulator, it is possible to simulate directly any cache size, placement, fetch or replacement algorithm, block size, etc. This type of simulation has become the mainstay of memory hierarchy evaluation for the last 20 or so years and is widely accepted in the research community [SMIT82], [HILL89], [POUR94]. In fact, in many cases, trace-driven simulation is preferred to any type of actual measurement since these simulations are repeatable and allow cache parameters to be varied so that effects can be isolated. They are cheaper than actual hardware monitoring and do not require access to, or the existence of, the machine being studied.

D.2 Program Tracing

Since trace-driven simulation is used to evaluate performance of cache configurations, address traces of selected benchmark programs are required. Traces can be generated by several methods. *Hardware-captured traces* are captured by hardware performance monitor logic that directly records physical memory references. Complexity, cost, and lack of flexibility are the primary limitations of this approach. *Interrupt-based traces* are generated when the

program is interrupted after execution of each instruction, and address information is recorded. The problem with this method is that the need for interrupting every instruction slows down the trace generation significantly. *Simulation-based traces* are based on the use of a cycle-level architectural simulator that executes the simulated machine at the binary level. By implementing the "programmer's model" of the machine exactly, address can be easily traced. However, this method requires extensive coding of specific architectures, and is not easily modified. *Microcode-based traces* are generated by modify microcode to record instruction and data addresses. While this method is generally very fast, RISC processors in general do not contain microcode and current CISC processors have their microcode in their read-only memory (ROM) which is not modifiable (The ATUM project [AGAR86] made this method popular by generating many traces for the DEC VAX-11/780 processor). In *instrumented program-based traces*, the program source code (operating system or application) is directly modified for trace generation. By interspersing instructions to record address information at strategic locations in the existing code, address traces can be efficiently generated [POUR94]. This method has attracted considerable attention in recent years, and was chosen for this research due to its flexibility and efficiency. The *Quick Profiling and Tracing System* (QPT), written by James Larus [LARU93],

[BALL92], was chosen to generate the require traces due to its flexibility, efficiency, interface to cache simulators, and widespread use in the research community [LARU92], [LEBE94], [LIU93].

The QPT tracing program is an exact and efficient program tracing system and is available as part of the *Wisconsin Architectural Research Tools Set (WARTS)* [HILL93].[27] The QPT tools rewrites a program's executable file (*a.out*) by inserting code into the file to record the execution sequence of every basic block (straight-line sequence of instructions) or control-flow edge (conditional execution). From this information, QPT regenerates a full program trace on instruction and data references on demand [LARU93]. Figure D.1 illustrates the steps required to generate program traces in cache simulator readable format. QPT adds tracing code to a program's executable file (*a.out*) and produces the traced application (*a.out.qpt*) and a trace generation program (*a.out_sma.c*). The latter program is linked to an application program (i.e., *din.c*) that writes the program trace is a format suitable for the cache simulator being used.

---

[27] QPT also profiles programs. For example, it generates program statistics such as the frequency of specific basic block execution.

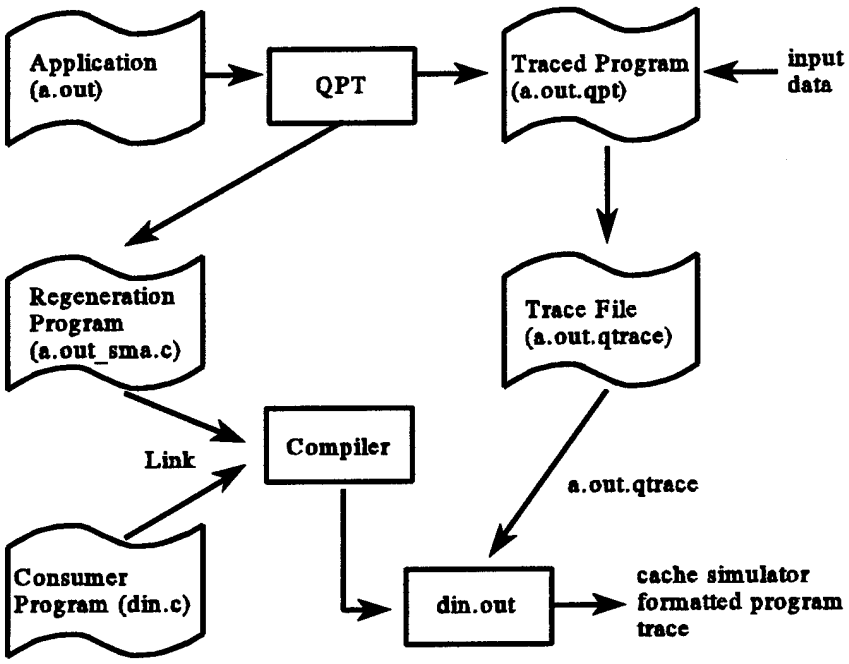


Figure D.1 - QPT Program Tracing Flow Chart [LARU93]
(©1993 IEEE, Reprinted with Permission)

D.3 Cache Simulator

The cache simulator chosen to simulate the cache configuration is *DineroIII* written by Mark Hill. It was selected due to its widespread use and availability [KIRK90], [LIU93]. The simulator reports the behavior of one or more alternative cache designs in response to an input program trace provided by the user (e.g, with QPT) and specified cache parameters. Cache parameters (e.g., block size, associativity, etc.) are set with command line options [DINE94]. A unified cache

(instructions and data cache together) or split cache (separate instruction and data caches) can be simulated. Several parameters can be varied, and are listed in Table D.1. The default parameter settings are listed in the footnotes to the table.

Table D.1 - Possible Cache Simulator Parameter Configurations

		Variable Cache Parameters									
		Cache Size [1]	Unified/ Split	Block Size [1]	Sub-block Size [2]	Assoc [3]	Replacement Policy [4]	Fetch Policy [5]	Prefetch Distance [6]	Write Policy [7]	Allocate Policy [8]
Fixed Cache Parameters	Cache Size [1]	n/a									
	Unified/Split		n/a								
	Block Size [1]			n/a							
	Sub-block Size [2]				n/a						
	Associativity [3]					n/a					
	Replacement Policy [4]						n/a				
	Fetch Policy [5]							n/a			
	Prefetch Distance [6]								n/a		
	Write Policy [7]									n/a	
	Allocate Policy [8]										n/a

Footnotes to Table D.1 (default simulator settings for *DineroIII* in **bold**)

1 - size in bytes; **no default**

2 - size in bytes; **0 (no sub-block)**

3 - n-way associativity; **1 (direct mapped)**

4 - **LRU**, FIFO, or Random

5 - **demand fetch (no prefetching)**; always-prefetch; prefetch after demand miss; tagged prefetch; load-forward prefetch; sub-block prefetch

6 - prefetch distance in sub-blocks (if enabled) or blocks otherwise; **1 (sub) block**

7 - write-through or **copy-back**

8 - **write allocate** or no-write allocate

Appendix E Simulation Results

E.1 IMD=0, 1, and 2 for SHUTTLE.c Benchmark

Table E.1 - IMD=0 Count, Cache Size=8k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	512	240	165	146	147	333	137	119	112	114
32	185	112	77	64	69	65	54	41	38	34
64	71	36	36	38	34	29	21	17	13	14
128	125	26	28	22	20	17	15	12	11	10
256	140	20	35	36	37	6	3	4	4	3
512	619	98	73	36	42	4	1	1	2	1
1024	2446	3	1	1	n/a	6	0	0	0	n/a

Table E.2 - IMD=0 Count, Cache Size=16k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	267	119	107	100	99	178	97	97	96	95
32	115	49	44	37	34	38	32	35	31	30
64	28	21	21	16	17	15	11	12	10	10
128	43	11	9	11	11	7	6	5	7	5
256	71	4	2	4	4	3	1	1	1	1
512	510	5	0	2	2	3	0	0	0	0
1024	2312	0	0	0	0	6	0	0	0	0

Table E.3 - IMD=0 Count, Cache Size=32k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	153	102	95	95	95	100	95	95	95	95
32	76	35	30	30	30	34	30	30	30	30
64	20	12	9	9	9	11	9	9	9	9
128	10	7	4	4	4	4	4	4	4	4
256	35	3	1	1	1	0	0	0	0	0
512	264	1	0	0	0	0	0	0	0	0
1024	230	0	0	0	0	0	0	0	0	0

Table E.4 - IMD=0 Count, Cache Size=64k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	108	96	95	95	95	95	95	95	95	95
32	37	31	30	30	30	30	30	30	30	30
64	17	10	9	9	9	9	9	9	9	9
128	9	4	4	4	4	4	4	4	4	4
256	4	0	0	0	0	0	0	0	0	0
512	6	0	0	0	0	0	0	0	0	0
1024	6	0	0	0	0	0	0	0	0	0

Table E.5 - IMD=0 Count, Cache Size=128k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	96	95	95	95	95	95	95	95	95	95
32	30	30	30	30	30	30	30	30	30	30
64	9	9	9	9	9	9	9	9	9	9
128	4	4	4	4	4	4	4	4	4	4
256	0	0	0	0	0	0	0	0	0	0
512	0	0	0	0	0	0	0	0	0	0
1024	0	0	0	0	0	0	0	0	0	0

Table E.6 - IMD=1 Count, Cache Size=8k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	561	338	198	184	173	339	181	150	151	154
32	262	150	103	94	88	175	86	75	71	69
64	183	141	71	70	71	73	49	42	40	34
128	108	44	40	34	32	36	18	13	14	15
256	100	40	31	25	24	29	12	8	8	8
512	153	53	60	97	83	51	10	1	2	3
1024	289	82	68	36	n/a	94	1	3	3	n/a

Table E.7 - IMD=1 Count, Cache Size=16k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	465	156	148	129	124	271	138	132	121	121
32	165	70	69	59	63	97	64	62	54	53
64	106	39	36	34	35	33	32	31	24	26
128	60	11	10	10	11	18	8	8	7	8
256	53	6	6	7	8	16	5	4	5	5
512	116	2	1	1	2	34	1	1	1	1
1024	208	8	2	2	2	60	1	1	1	1

Table E.8 - IMD=1 Count, Cache Size=32k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	314	131	121	120	120	122	121	121	121	121
32	121	57	53	52	53	54	53	53	53	53
64	89	28	23	23	23	25	24	24	24	24
128	28	7	6	6	6	7	7	7	7	7
256	30	4	4	4	4	4	4	5	4	3
512	63	1	1	1	1	1	1	1	1	1
1024	41	6	1	1	1	1	1	1	1	1

Table E.9 - IMD=1 Count, Cache Size=64k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	126	120	120	120	120	121	121	121	121	121
32	57	52	52	52	52	53	53	53	53	53
64	26	23	23	23	23	24	24	24	24	24
128	8	6	6	6	6	7	7	7	7	7
256	4	3	4	3	3	3	3	3	3	3
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.10 - IMD=1 Count, Cache Size=128k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	123	120	120	120	120	121	121	121	121	121
32	54	52	52	52	52	53	53	53	53	53
64	24	23	23	23	23	24	24	24	24	24
128	7	6	6	6	6	7	7	7	7	7
256	4	3	3	3	3	3	3	3	3	3
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.11 - IMD=2 Count, Cache Size=8k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	645	395	243	236	219	420	230	204	197	198
32	338	165	113	95	90	250	97	77	67	68
64	204	80	61	65	49	130	40	27	28	26
128	170	95	71	42	33	111	27	18	16	14
256	85	36	21	32	21	76	16	10	9	9
512	163	40	25	57	76	81	21	6	7	6
1024	326	184	127	109	n/a	89	55	9	40	n/a

Table E.12 - IMD=2 Count, Cache Size=16k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	466	197	192	166	166	273	171	169	158	158
32	144	78	70	65	62	100	59	57	53	53
64	45	41	35	26	27	27	21	19	20	19
128	33	34	31	11	12	20	8	8	8	9
256	20	8	7	6	8	15	5	5	5	6
512	100	14	13	7	6	18	4	4	5	5
1024	240	16	20	6	40	18	4	5	5	5

Table E.13 - IMD=2 Count, Cache Size=32k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	351	168	159	157	157	159	158	158	158	158
32	87	58	53	54	53	53	53	53	53	53
64	22	22	19	19	19	19	19	19	19	19
128	11	9	7	7	7	9	8	8	8	8
256	4	4	4	4	4	5	5	5	5	5
512	5	6	4	4	4	6	4	4	4	4
1024	11	5	3	4	3	6	4	4	4	4

Table E.14 - IMD=2 Count, Cache Size=64k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	164	157	157	157	157	157	157	157	157	157
32	55	53	53	53	53	53	53	53	53	53
64	21	19	19	19	19	19	19	19	19	19
128	10	7	7	7	7	8	8	8	8	8
256	5	4	4	4	4	4	4	4	4	4
512	9	6	4	4	4	4	4	4	4	4
1024	11	5	3	3	3	4	4	4	4	4

Table E.15 - IMD=2 Count, Cache Size=128k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	161	157	157	157	157	158	158	158	158	158
32	54	53	53	53	53	53	53	53	53	53
64	19	19	19	19	19	19	19	19	19	19
128	7	7	7	7	7	8	8	8	8	8
256	4	4	4	4	4	5	5	5	5	5
512	4	4	4	4	4	4	4	4	4	4
1024	3	3	3	3	3	4	4	4	4	4

Table E.16 - IMD=0 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	268	112	104	105	108	125	109	98	97	97
32	30	28	27	27	27	46	29	12	10	10
64	6	6	6	6	6	46	27	9	7	7
128	5	6	4	4	4	94	49	32	43	43
256	1	1	1	1	1	116	72	28	25	29
512	0	0	0	0	0	229	55	42	14	14
1024	0	0	0	0	n/a	379	155	51	21	n/a

Table E.17 - IMD=0 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	129	85	85	83	82	112	97	97	97	97
32	26	24	24	23	22	24	10	10	10	11
64	5	4	4	4	4	18	4	4	4	4
128	3	3	3	3	3	61	4	6	3	4
256	1	1	1	1	1	90	4	3	4	3
512	0	0	0	0	0	199	3	2	2	3
1024	0	0	0	0	0	363	4	3	3	3

Table E.18 - IMD=0 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	82	82	82	82	82	98	97	97	97	97
32	22	22	22	22	22	13	11	11	11	11
64	3	3	3	3	3	6	4	4	4	4
4128	2	2	2	2	2	22	3	3	3	3
256	0	0	0	0	0	30	4	3	3	3
512	0	0	0	0	0	17	3	2	2	2
1024	0	0	0	0	0	27	4	2	2	2

Table E.19 - IMD=0 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	82	82	82	82	82	97	97	97	97	97
32	22	22	22	22	22	11	11	11	11	11
64	3	3	3	3	3	5	4	4	4	3
128	2	2	2	2	2	3	3	3	3	3
256	0	0	0	0	0	3	3	3	3	3
512	0	0	0	0	0	2	2	2	2	2
1024	0	0	0	0	0	3	2	2	2	2

Table E.20 - IMD=0 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	82	82	82	82	82	97	97	97	97	97
32	22	22	22	22	22	11	11	11	11	11
64	3	3	3	3	3	5	4	4	4	4
128	2	2	2	2	2	3	3	3	3	3
256	0	0	0	0	0	3	3	3	3	3
512	0	0	0	0	0	2	2	2	2	2
1024	0	0	0	0	0	2	2	2	2	2

Table E.21 - IMD=1 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	302	157	132	132	136	16	15	14	12	12
32	145	74	66	60	60	87	82	54	50	50
64	80	40	37	37	31	47	45	12	12	12
128	39	11	9	7	9	59	54	46	12	13
256	3	2	2	2	2	43	55	44	24	31
512	1	0	0	1	1	67	91	71	31	13
1024	1	0	2	2	n/a	193	84	40	40	n/a

Table E.22 - IMD=1 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	218	116	112	102	102	12	12	12	12	12
32	54	51	49	41	41	82	50	50	50	50
64	28	27	26	20	21	43	10	11	10	10
128	6	6	6	5	6	55	7	6	9	7
256	1	1	1	1	1	34	9	10	8	6
512	0	0	0	0	0	60	7	6	6	5
1024	0	0	0	0	0	171	5	4	4	5

Table E.23 - IMD=1 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	102	102	102	102	102	12	12	12	12	12
32	41	41	41	41	41	50	50	50	50	50
64	19	19	19	19	19	10	10	10	10	10
128	5	5	5	5	5	17	6	6	7	6
256	1	1	1	1	1	14	8	5	5	5
512	0	0	0	0	0	26	7	4	4	4
1024	0	0	0	0	0	5	5	2	2	2

Table E.24 - IMD=1 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	102	102	102	102	102	12	12	12	12	12
32	41	41	41	41	41	50	50	50	50	50
64	19	19	19	19	19	10	10	10	10	10
128	5	5	5	5	5	6	6	6	6	6
256	1	1	1	1	1	6	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.25 - IMD=1 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	102	102	102	102	102	12	12	12	12	12
32	41	41	41	41	41	50	50	50	50	50
64	19	19	19	19	19	10	10	10	10	10
128	5	5	5	5	5	6	6	6	6	6
256	1	1	1	1	1	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.26 - IMD=2 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	392	207	192	194	193	19	14	15	12	12
32	218	67	58	57	57	18	13	10	9	9
64	102	49	16	15	14	24	18	14	13	15
128	66	36	8	7	5	14	10	12	4	5
256	33	3	3	1	2	26	5	25	8	31
512	33	3	3	3	3	82	23	47	22	10
1024	33	2	2	2	n/a	98	18	4	4	n/a

Table E.27 - IMD=2 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	280	155	153	143	141	20	14	12	12	12
32	110	45	46	42	40	18	10	9	9	9
64	43	10	10	10	10	23	14	12	12	12
128	34	2	2	2	2	12	3	3	3	2
256	1	1	1	1	1	18	2	2	2	2
512	1	1	1	1	1	67	2	3	4	3
1024	1	1	1	1	1	80	2	2	2	2

Table E.28 - IMD=2 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: SHUTTLE.c

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	141	141	141	141	141	12	12	12	12	12
32	40	40	40	40	40	9	9	9	9	9
64	10	10	10	10	10	13	13	13	13	13
128	2	2	2	2	2	3	2	2	2	2
256	1	1	1	1	1	1	1	1	1	1
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	9	1	1	1	1

Table E.29 - IMD=2 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	141	141	141	141	141	12	12	12	12	12
32	40	40	40	40	40	9	9	9	9	9
64	10	10	10	10	10	13	13	13	13	13
128	2	2	2	2	2	2	2	2	2	2
256	1	1	1	1	1	1	1	1	1	1
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.30 - IMD=2 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: SHUTTLE.c										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	141	141	141	141	141	12	12	12	12	12
32	40	40	40	40	40	9	9	9	9	9
64	10	10	10	10	10	13	13	13	13	13
128	2	2	2	2	2	2	2	2	2	2
256	1	1	1	1	1	1	1	1	1	1
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

E.2 IMD=0, 1, and 2 for LRCpr1.c.ok Benchmark

Table E.31 - IMD=0 Count, Cache Size=8k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	215	162	107	98	119	142	100	8	83	80
32	152	88	72	49	48	72	51	42	38	36
64	105	23	13	20	14	20	9	9	10	9
128	145	7	8	7	11	4	3	2	2	2
256	258	5	9	5	11	2	1	2	2	1
512	443	4	4	4	11	1	1	1	2	2
1024	468	11	11	11	n/a	12	10	1	1	n/a

Table E.32 - IMD=0 Count, Cache Size=16k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	141	113	83	115	115	104	83	80	80	80
32	82	55	38	34	33	53	34	33	33	33
64	35	17	8	8	8	7	7	7	7	7
128	35	2	1	2	3	1	1	1	1	1
256	71	1	1	2	2	0	0	0	0	0
512	118	1	1	1	1	0	1	1	1	1
1024	125	2	3	1	1	0	0	0	0	0

Table E.33 - IMD=0 Count, Cache Size=32k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	112	95	78	78	81	80	80	80	80	80
32	57	45	45	45	45	34	33	33	33	33
64	17	7	7	7	7	7	7	7	7	7
128	2	1	1	1	1	1	1	1	1	1
256	2	0	0	0	0	0	0	0	0	0
512	12	0	0	0	0	0	0	0	0	0
1024	12	0	0	0	0	0	0	0	0	0

Table E.34 - IMD=0 Count, Cache Size=64k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	80	78	78	78	78	81	80	80	80	80
32	35	33	33	33	33	34	33	33	33	33
64	7	7	7	7	7	7	7	7	7	7
128	1	1	1	1	1	1	1	1	1	1
256	1	0	0	0	0	0	0	0	0	0
512	0	0	0	0	0	0	0	0	0	0
1024	0	0	0	0	0	0	0	0	0	0

Table E.35 - IMD=0 Count, Cache Size=128k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	79	78	78	78	78	81	80	80	80	80
32	45	33	33	33	33	34	33	33	33	33
64	7	7	7	7	7	7	7	7	7	7
128	1	1	1	1	1	1	1	1	1	1
256	0	0	0	0	0	0	0	0	0	0
512	0	0	0	0	0	0	0	0	0	0
1024	0	0	0	0	0	0	0	0	0	0

Table E.36 - IMD=1 Count, Cache Size=8k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	269	227	154	132	120	185	154	125	117	117
32	115	98	85	59	51	64	66	59	49	48
64	69	56	52	34	36	32	41	34	27	24
128	44	15	27	12	15	15	8	8	8	8
256	39	11	6	12	7	8	4	4	3	3
512	50	15	13	13	13	5	4	3	2	3
1024	42	25	3	3	n/a	3	1	3	3	n/a

Table E.37 - IMD=1 Count, Cache Size=16k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	186	152	117	116	116	145	118	117	117	117
32	77	54	46	46	46	49	46	46	46	46
64	31	22	23	22	22	23	22	22	22	22
128	12	5	6	6	6	7	5	5	5	5
256	11	3	4	4	3	2	2	2	2	2
512	10	4	4	2	3	1	1	1	1	2
1024	6	2	1	3	3	1	1	1	1	1

Table E.38 - IMD=1 Count, Cache Size=32k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	146	126	116	116	116	117	117	117	117	117
32	58	46	46	46	46	46	46	46	46	46
64	26	22	22	22	22	22	22	22	22	22
128	6	5	5	5	5	5	5	5	5	5
256	3	2	2	2	2	2	2	2	2	2.
512	3	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.39 - IMD=1 Count, Cache Size=64k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	119	116	116	116	116	117	117	117	117	117
32	49	46	46	46	46	46	46	46	46	46
64	26	22	22	22	22	22	22	22	22	22
128	6	5	5	5	5	5	5	5	5	5
256	3	2	2	2	2	2	2	2	2	2
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.40 - IMD=1 Count, Cache Size=128k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	116	116	116	116	116	117	117	117	117	117
32	46	46	46	46	46	46	46	46	46	46
64	22	22	22	22	22	22	22	22	22	22
128	5	5	5	5	5	5	5	5	5	5
256	2	2	2	2	2	2	2	2	2	2
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.41 - IMD=2 Count, Cache Size=8k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	297	225	147	133	126	196	147	118	117	116
32	148	112	85	56	53	75	52	54	44	41
64	77	56	32	24	24	28	20	19	17	15
128	74	29	14	12	16	22	12	10	10	12
256	82	25	17	13	9	39	8	7	5	6
512	76	20	7	7	7	30	8	7	9	8
1024	85	16	5	5	n/a	29	6	7	16	n/a

Table E.42 - IMD=2 Count, Cache Size=16k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	230	167	120	119	117	164	118	116	116	116
32	108	68	42	40	40	61	41	41	41	41
64	41	23	14	13	13	17	14	14	14	14
128	45	16	9	8	7	9	8	8	8	8
256	41	15	6	6	6	5	5	5	5	5
512	44	17	6	5	6	5	5	5	5	5
1024	44	15	3	3	4	4	4	4	4	4

Table E.43 - IMD=2 Count, Cache Size=32k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	201	135	117	117	117	117	116	116	116	116
32	78	49	40	40	40	41	41	41	41	41
64	15	14	13	13	13	14	14	14	14	14
128	7	7	7	7	7	8	8	8	8	8
256	5	4	4	4	4	5	5	5	5	5
512	4	4	4	4	4	5	5	5	5	5
1024	2	2	2	2	2	4	4	4	4	4

Table E.44 - IMD=2 Count, Cache Size=64k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	120	117	117	117	117	117	116	116	116	116
32	42	40	40	40	40	41	41	41	41	41
64	14	13	13	13	13	14	14	14	14	14
128	7	7	7	7	7	8	8	8	8	8
256	5	4	4	4	4	5	5	5	5	5
512	4	4	4	4	4	5	5	5	5	5
1024	2	2	2	2	2	4	4	4	4	4

Table E.45 - IMD=2 Count, Cache Size=128k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	118	117	117	117	117	117	116	116	116	116
32	40	40	40	40	40	41	41	41	41	41
64	13	13	13	13	13	14	14	14	14	14
128	7	7	7	7	7	8	8	8	8	8
256	4	4	4	4	4	5	5	5	5	5
512	4	4	4	4	4	5	5	5	5	5
1024	2	2	2	2	2	4	4	4	4	4

Table E.46 - IMD=0 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	110	83	71	67	64	104	98	98	98	99
32	53	42	34	31	29	13	10	10	10	10
64	16	6	6	7	6	11	4	4	4	4
128	6	4	3	3	3	11	5	4	4	4
256	1	1	1	1	1	24	6	4	4	4
512	1	1	1	1	1	32	8	3	5	6
1024	1	0	0	0	n/a	60	52	4	5	n/a

Table E.47 - IMD=0 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	80	66	64	64	64	100	99	99	99	99
32	38	26	26	26	26	11	11	11	11	11
64	5	4	4	4	4	4	4	4	4	4
128	3	2	2	2	2	4	4	4	4	4
256	0	0	0	0	0	4	4	4	4	4
512	0	1	1	1	1	13	3	3	3	3
1024	0	0	0	0	0	22	22	3	2	2

Table E.48 - IMD=0 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	64	64	64	64	64	100	99	99	99	99
32	26	26	26	26	26	11	11	11	11	11
64	4	4	4	4	4	4	4	4	4	4
128	2	2	2	2	2	4	4	4	4	4
256	0	0	0	0	0	4	4	4	4	4
512	0	0	0	0	0	3	3	3	3	3
1024	0	0	0	0	0	3	3	3	2	2

Table E.49 - IMD=0 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	64	64	64	64	64	100	99	99	99	99
32	26	26	26	26	26	11	11	11	11	11
64	4	4	4	4	4	4	4	4	4	4
128	2	2	2	2	2	4	4	4	4	4
256	0	0	0	0	0	4	4	4	4	4
512	0	0	0	0	0	3	3	3	3	3
1024	0	0	0	0	0	2	3	2	2	2

Table E.50 - IMD=0 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	64	64	64	64	64	100	99	99	9	99
32	26	26	26	26	26	11	11	11	11	11
64	4	4	4	4	4	4	4	4	4	4
128	2	2	2	2	2	4	4	4	4	4
256	0	0	0	0	0	4	4	4	4	4
512	0	0	0	0	0	3	3	3	3	3
1024	0	0	0	0	0	2	2	2	2	2

Table E.51 - IMD=1 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	184	151	112	104	104	11	9	9	9	9
32	51	59	59	39	38	57	51	50	50	50
64	16	33	25	18	15	8	8	8	8	8
128	6	5	5	6	6	6	8	6	6	6
256	3	1	2	1	1	15	8	6	6	5
512	2	1	1	1	1	34	38	15	7	7
1024	0	0	1	1	n/a	22	31	13	13	n/a

Table E.52 - IMD=1 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	141	104	104	104	104	10	9	9	9	9
32	40	36	36	36	36	52	50	50	50	50
64	13	12	12	12	12	8	8	8	8	8
128	3	1	1	1	1	6	6	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	1	4	4	4	5	4
1024	0	0	0	0	0	2	2	3	2	2

Table E.53 - IMD=1 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	104	104	104	104	104	9	9	9	9	9
32	36	36	36	36	36	50	50	50	50	50
64	12	12	12	12	12	8	8	8	8	8
128	1	1	1	1	1	6	6	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.54 - IMD=1 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	104	104	104	104	104	9	9	9	9	9
32	36	36	36	36	36	51	50	50	50	50
64	12	12	12	12	12	8	8	8	8	8
128	1	1	1	1	1	6	6	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.55 - IMD=1 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	104	104	104	104	104	9	9	9	9	9
32	36	36	36	36	36	51	50	50	50	50
64	12	12	12	12	12	8	8	8	8	8
128	1	1	1	1	1	6	6	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.56 - IMD=2 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: LRCpr1.c.ok										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	183	137	116	113	111	11	4	3	3	3
32	61	38	33	33	30	14	5	4	4	4
64	20	14	22	9	8	16	8	7	7	7
128	14	5	6	4	6	7	2	1	1	1
256	11	3	3	2	3	14	1	3	1	1
512	11	2	2	2	2	14	3	2	1	1
1024	11	2	2	2	n/a	24	33	1	1	n/a

Table E.57 - IMD=2 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	161	112	111	111	111	4	4	3	3	3
32	49	30	30	30	30	5	5	4	4	4
64	9	8	7	7	7	9	9	8	8	8
128	3	3	3	3	3	3	2	1	1	1
256	2	2	2	2	2	3	2	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	3	2	2	2	1

Table E.58 - IMD=2 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	111	111	111	111	111	4	3	3	3	3
32	30	30	30	30	30	4	4	4	4	4
64	7	7	7	7	7	9	8	8	8	8
128	3	3	3	3	3	2	1	1	1	1
256	2	2	2	2	2	2	1	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	2	2	2	1	1

Table E.59 - IMD=2 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	111	111	111	111	111	4	3	3	3	3
32	30	30	30	30	30	5	4	4	4	4
64	7	7	7	7	7	9	8	8	8	8
128	3	3	3	3	3	2	1	1	1	1
256	2	2	2	2	2	2	1	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	2	2	1	1	1

Table E.60 - IMD=2 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: LRCpr1.c.ok

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	111	111	111	111	111	4	3	3	3	3
332	30	30	30	30	30	5	4	4	4	4
64	7	7	7	7	7	9	8	8	8	8
128	3	3	3	3	3	2	1	1	1	1
256	2	2	2	2	2	2	1	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	2	1	1	1	1

E.3 IMD=0, 1, and 2 for LRCpr1.c.fail Benchmark

Table E.61 - IMD=0 Count, Cache Size=8k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	265	187	134	131	128	178	119	109	100	97
32	182	100	77	60	58	90	56	48	44	41
64	131	26	19	26	19	30	12	13	13	13
128	240	14	11	9	12	8	5	5	3	4
256	372	7	11	6	12	3	2	3	3	2
512	900	6	7	3	11	1	1	1	2	2
1024	1470	34	17	17	n/a	13	10	1	1	n/a

Table E.62 - IMD=0 Count, Cache Size=16k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	169	129	96	90	88	126	96	90	90	90
32	99	59	38	34	38	63	37	35	36	35
64	55	22	10	11	12	11	9	8	9	8
128	120	5	2	4	4	4	3	2	3	2
256	162	2	2	3	3	1	1	1	2	1
512	478	3	1	1	1	0	1	1	1	1
1024	963	25	5	1	1	1	0	0	0	0

Table E.63 - IMD=0 Count, Cache Size=32k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	121	108	89	88	88	91	90	90	90	90
32	48	53	35	35	35	36	35	35	35	35
64	17	8	8	8	8	8	8	8	8	8
128	3	2	2	2	2	2	2	2	2	2
256	3	1	1	1	1	1	1	1	1	1
512	23	2	0	0	0	0	0	0	0	0
1024	23	23	0	0	0	0	0	0	0	0

Table E.64 - IMD=0 Count, Cache Size=64k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	91	88	88	88	88	91	90	90	90	90
32	37	35	35	35	35	36	35	35	35	35
64	8	8	8	8	8	8	8	8	8	8
128	2	2	2	2	2	2	2	2	2	2
256	2	1	1	1	1	1	1	1	1	1
512	2	0	0	0	0	0	0	0	0	0
1024	2	0	0	0	0	0	0	0	0	0

Table E.65 - IMD=0 Count, Cache Size=128k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	90	88	88	88	88	91	90	90	90	90
32	36	35	35	35	35	36	35	35	35	35
64	8	8	8	8	8	8	8	8	8	8
128	2	2	2	2	2	2	2	2	2	2
256	1	1	1	1	1	1	1	1	1	1
512	2	0	0	0	0	0	0	0	0	0
1024	2	0	0	0	0	0	0	0	0	0

Table E.66 - IMD=1 Count, Cache Size=8k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	317	263	190	176	170	230	182	155	149	146
32	140	106	92	69	66	88	71	64	55	56
64	111	63	57	45	44	64	48	38	31	29
128	115	32	21	19	22	61	18	10	12	10
256	194	21	8	16	10	152	5	4	3	4
512	253	26	18	17	17	149	5	4	3	4
1024	510	34	6	7	n/a	357	3	3	3	n/a

Table E.67 - IMD=1 Count, Cache Size=16k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	220	181	145	136	133	176	138	136	134	134
32	98	60	52	49	47	68	48	48	47	47
64	70	28	30	26	26	52	25	24	24	24
128	66	11	9	9	8	47	10	8	7	7
256	159	5	4	5	4	144	2	2	2	2
512	197	8	3	3	4	143	1	1	1	1
1024	423	4	1	3	3	354	1	1	1	1

Table E.68 - IMD=1 Count, Cache Size=32k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	165	146	133	133	133	134	134	134	134	134
32	59	48	47	47	47	47	47	47	47	47
64	29	26	24	24	24	24	24	24	24	24
128	8	7	7	8	7	7	7	7	7	7
256	3	2	2	2	2	2	2	2	2	2
512	4	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.69 - IMD=1 Count, Cache Size=64k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	137	133	133	133	133	134	134	134	134	134
32	51	47	47	47	47	47	47	47	47	47
64	29	24	24	24	24	24	24	24	24	24
128	8	7	7	7	7	7	7	7	7	7
256	3	2	2	2	2	2	2	2	2	2
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.70 - IMD=1 Count, Cache Size=128k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	133	133	133	133	133	134	134	134	134	134
32	47	47	47	47	47	47	47	47	47	47
64	24	24	24	24	24	24	24	24	24	24
128	7	7	7	7	7	7	7	7	7	7
256	2	2	2	2	2	2	2	2	2	2
512	1	1	1	1	1	1	1	1	1	1
1024	1	1	1	1	1	1	1	1	1	1

Table E.71 - IMD=2 Count, Cache Size=8k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	350	294	177	168	170	245	174	148	139	137
32	176	126	99	76	72	97	63	65	59	55
64	102	67	35	31	31	38	23	23	24	22
128	109	38	16	17	21	41	16	11	12	14
256	122	30	20	12	12	69	10	8	6	7
512	157	30	8	10	8	59	9	8	11	9
1024	276	32	5	7	n/a	50	6	7	16	n/a

Table E.72 - IMD=2 Count, Cache Size=16k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	264	195	142	134	131	194	133	131	130	130
32	125	81	52	48	47	74	49	47	47	47
64	59	30	19	21	19	22	19	18	17	17
128	64	20	10	9	9	23	11	9	9	9
256	69	16	8	7	6	23	7	6	6	6
512	108	26	6	6	6	22	6	6	5	5
1024	209	24	3	3	4	10	4	4	4	4

Table E.73 - IMD=2 Count, Cache Size=32k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	217	157	131	131	131	131	130	130	130	130
32	83	59	46	46	46	47	47	47	47	47
64	17	16	16	16	16	17	17	17	17	17
128	8	9	8	8	8	9	9	9	9	9
256	7	6	6	5	5	7	7	6	6	6
512	5	5	5	4	4	6	6	5	5	5
1024	2	2	2	2	2	4	4	4	4	4

Table E.74 - IMD=2 Count, Cache Size=64k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	134	131	131	131	131	131	130	130	130	130
32	48	46	46	46	46	47	47	47	47	47
64	17	16	16	16	16	17	17	17	17	17
128	8	8	8	8	8	9	9	9	9	9
256	7	5	5	5	5	7	6	6	6	6
512	5	4	4	4	4	6	5	5	5	5
1024	2	2	2	2	2	4	4	4	4	4

Table E.75 - IMD=2 Count, Cache Size=128k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Unified Cache					Split Cache				
	1	2	4	8	16	1	2	4	8	16
16	134	131	131	131	131	131	130	130	130	130
32	47	46	46	46	46	47	47	47	47	47
64	17	16	16	16	16	17	17	17	17	17
128	8	8	8	8	8	9	9	9	9	9
256	6	5	5	5	5	7	6	6	6	6
512	5	4	4	4	4	6	5	5	5	5
1024	2	2	2	2	2	4	4	4	4	4

Table E.76 - IMD=0 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	138	102	95	87	82	116	98	98	98	98
32	63	47	40	38	35	31	10	10	10	10
64	20	9	11	11	11	37	4	4	4	4
128	8	6	6	5	6	64	5	4	4	4
256	2	2	2	2	2	197	6	5	4	4
512	1	1	1	1	1	205	8	4	6	7
1024	1	0	0	0	n/a	553	51	4	5	n/a

Table E.77 - IMD=0 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	95	78	73	73	73	111	99	99	99	99
32	42	28	27	28	27	28	11	11	10	11
64	7	6	5	6	5	31	4	4	4	4
128	5	4	3	4	3	57	4	4	4	4
256	1	1	1	2	1	168	4	4	4	4
512	0	1	1	1	1	177	3	3	3	3
1024	0	0	0	0	0	506	22	3	2	2

Table E.78 - IMD=0 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	73	73	73	73	73	102	99	99	99	99
32	27	27	27	27	27	11	11	11	11	11
64	3	3	3	3	3	4	4	4	4	4
128	1	1	1	1	1	4	4	4	4	4
256	1	1	1	1	1	4	4	4	4	4
512	0	0	0	0	0	3	3	3	3	3
1024	0	0	0	0	0	3	3	3	2	2

Table E.79 - IMD=0 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	73	73	73	73	73	102	99	99	99	99
32	27	27	27	27	27	11	11	11	11	11
64	5	5	5	5	5	4	4	4	4	4
128	3	3	3	3	3	4	4	4	4	4
256	1	1	1	1	1	4	4	4	4	4
512	0	0	0	0	0	3	3	3	3	3
1024	0	0	0	0	0	2	3	3	3	2

Table E.80 - IMD=0 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	73	73	73	73	73	102	99	99	99	99
32	27	27	27	27	27	11	11	11	11	11
64	5	5	5	5	5	4	4	4	4	4
128	3	3	3	3	3	4	4	4	4	4
256	1	1	1	1	1	4	4	4	4	4
512	0	0	0	0	0	3	3	3	3	3
1024	0	0	0	0	0	2	2	2	2	2

Table E.81 - IMD=1 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	225	181	147	139	136	16	10	9	9	9
32	61	64	66	49	50	61	51	50	50	50
64	24	38	29	23	21	12	8	8	8	8
128	13	12	5	9	7	20	11	6	6	6
256	4	2	1	1	2	48	9	6	6	5
512	3	2	1	1	1	67	34	15	9	9
1024	0	1	1	1	n/a	115	30	13	13	n/a

Table E.82 - IMD=1 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	169	129	129	127	127	14	10	9	9	9
32	44	39	39	38	38	57	50	50	50	50
64	17	16	15	15	15	12	8	8	8	8
128	5	6	4	3	3	20	8	6	6	6
256	0	0	0	0	0	38	5	5	5	5
512	0	0	0	0	0	38	4	4	4	4
1024	0	0	0	0	0	95	6	4	2	2

Table E.83 - IMD=1 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	127	127	127	127	127	10	9	9	9	9
32	38	38	38	38	38	53	50	50	50	50
64	15	15	15	15	15	8	8	8	8	8
128	3	3	3	3	3	6	8	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	5	2	2	2

Table E.84 - IMD=1 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	127	127	127	127	127	10	9	9	9	9
32	38	38	38	38	38	53	50	50	50	50
64	15	15	15	15	15	8	8	8	8	8
128	3	3	3	3	3	6	6	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.85 - IMD=1 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	127	127	127	127	127	10	9	9	9	9
32	38	38	38	38	38	53	50	50	50	50
64	15	15	15	15	15	8	8	8	8	8
128	3	3	3	3	3	6	6	6	6	6
256	0	0	0	0	0	5	5	5	5	5
512	0	0	0	0	0	4	4	4	4	4
1024	0	0	0	0	0	2	2	2	2	2

Table E.86 - IMD=2 Count, Instruction/Data Caches, Cache Size=8k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	220	166	142	134	131	13	5	4	4	4
32	74	49	43	44	41	17	6	5	5	5
64	25	18	26	16	14	32	8	7	7	7
128	17	9	8	6	7	43	2	1	1	1
256	12	4	4	3	4	79	1	3	1	1
512	11	3	2	2	2	79	3	2	1	1
1024	11	2	2	2	n/a	184	32	1	1	n/a

Table E.87 - IMD=2 Count, Instruction/Data Caches, Cache Size=16k, Benchmark: LRCpr1.c.fail

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	181	122	121	120	120	7	5	4	4	4
32	57	37	36	36	36	8	6	5	5	5
64	11	12	11	10	10	25	9	8	7	8
128	4	5	4	4	4	39	2	1	1	1
256	3	3	3	3	3	68	2	1	1	1
512	1	1	1	1	1	68	2	1	1	1
1024	1	1	1	1	1	163	2	2	2	1

Table E.88 - IMD=2 Count, Instruction/Data Caches, Cache Size=32k, Benchmark: LRCpr1.c.fail

Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	120	120	120	120	120	5	4	4	4	4
32	36	36	36	36	36	6	5	5	5	5
64	10	10	10	10	10	9	8	8	8	8
128	4	4	4	4	4	2	1	1	1	1
256	3	3	3	3	3	2	1	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	2	2	2	1	1

Table E.89 - IMD=2 Count, Instruction/Data Caches, Cache Size=64k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	120	120	120	120	120	5	4	4	4	4
32	36	36	36	36	36	6	5	5	5	5
64	10	10	10	10	10	9	8	8	8	8
128	4	4	4	4	4	2	1	1	1	1
256	3	3	3	3	3	2	1	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	2	2	1	1	1

Table E.90 - IMD=2 Count, Instruction/Data Caches, Cache Size=128k, Benchmark: LRCpr1.c.fail										
Block Size (bytes)	Associativity									
	Instruction Cache					Data Cache				
	1	2	4	8	16	1	2	4	8	16
16	120	120	120	120	120	5	4	4	4	4
32	36	36	36	36	36	6	5	5	5	5
64	10	10	10	10	10	9	8	8	8	8
128	4	4	4	4	4	2	1	1	1	1
256	3	3	3	3	3	2	1	1	1	1
512	1	1	1	1	1	2	1	1	1	1
1024	1	1	1	1	1	2	1	1	1	1

Bibliography

[ABRA93]
Abraham, S.G., Sugumar, R.A., Rau, B.R., and Gupta, R., "Predictability of Load/Store Instruction Latencies," *Proceedings of the 26th International Symposium on Microarchitecture*, December 1993, pp. 139-152

[AGAR86]
Agarwal, A., Sites, R.L., and Horowitz, M., "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986

[ALPH94]
Alpha 21164 Microprocessor Hardware Reference Manual, Rev 4, Digital Equipment Corporation, 1994

[BAER91]
Baer, J.L. and Chen, T.F., "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Supercomputing 91*, 1991, pp. 176-186

[BAER95]
Baer, J.L., Lee, D., Grunwald, D., Calder, B., "Instruction Cache Fetch Policies for Speculative Execution," *Computer Architecture News*, Vol. 23, No. 2, May 1995, pp. 357-367

[BALL92]
Ball, T. and Larus, J.R., "Optimally Profiling and Tracing Programs," *ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL)*, January 1992, pp. 59-70

[BLAK79]
Blakeslee, T. R., *Digital Design with Standard MSI & LSI,* 2nd ed., Wiley-Interscience, New York, 1979

[CHEN95]
Chen, T.F. and Baer, J.L., "Effective Hardware-Based Data Prefetching

for High-Performance Processors," *IEEE Transactions on Computers*, Vol. 44, No. 5, May 1995, pp. 609-623

[CLOP34]
Clopper, C.J. and Pearson, E.S., "The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial," *Biometrika*, Vol. 26, May 1934, pp. 404-413

[CONT95]
Conte, T.M., Menezes, K.N., Mills, P.M., and Patel, B.A., "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Computer Architecture News*, Vol. 23, No. 2, May 1995, pp. 333-344

[DINE94]
"DineroIII Manual Page," Distributed with *Wisconsin Architectural Tool Set*, 1994

[EICK93]
Eickemeyer, R.J. and Vassiliadis, S., "A Load-Instruction Unit for Pipelined Processors," *IBM Journal of Research and Development*, Vol. 37, No. 4, July 1993, pp. 547-564

[EGLE94]
Egler, M., "Problems with Caches for Multitasking Applications," *Computer Architecture News*, Vol. 22, No. 4, September 1994, pp. 78-79

[GIND77]
Gindele, J.D., "Buffer Block Prefetching Methods," *IBM Technical Disclosure Bulletin*, Vol. 20, No. 2, July 1977, pp. 696-697

[HALA91]
Halang, W.A. and Stoyenko, A.D., *Constructing Real-Time Systems*, Kluwer Academic Publishers, Boston, 1991

[HAND89]
Hand, T., "Real Time Systems Need Predictability," *Computer Design RISC Supplement*, November 1989, pp. 57-59

[HARN82]
Harnett, D.L., *Statistical Methods*, 3rd ed., Addison-Wesley, Reading, MA, 1982

[HELL84]
Helley, J.J. Jr., "A Distributed Expert System for Space Shuttle Flight Control," Ph.D. Dissertation, University of California at Los Angeles, 1984

[HENN90]
Hennessey, J.L. and Patterson, D.A., *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, San Mateo, CA, 1990

[HILL88]
Hill, M.D., "A Case for Direct-Mapped Caches," *IEEE Computer*, Vol. 21, No. 12, December 1988, pp. 25-40

[HILL89]
Hill, M.D. and Smith, A.J., "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1612-1630

[HILL93]
Hill, M.D., Larus, J.R., Lebeck, A.R., Talluir, M., and Wood, D.A., "Wisconsin Architectural Tool Set," *Computer Architecture News*, Vol. 21, No. 4, June 1993, pp. 8-10

[KIRK89]
Kirk, D.B., "SMART (Strategic Memory Allocation for Real-Time) Cache Design," *Proceedings of the Real-Time Systems Symposium*, December 1989, pp. 229-237

[KIRK90]
Kirk, D.B., "Predictable Cache Design for Real-Time Systems," Ph.D. Dissertation, Carnegie Mellon University, December 1990

[KIRK91]
Kirk, D. B., Strosnider, J. K., and Sasinowski, J. E., "Allocating SMART

Cache Segments for Schedulability," *Foundations of Real-Time Computing Scheduling and Resource Management*, A. van Tilborg and G. Koob Ed., Kluwer, 1991, pp. 251-276

[KLAI91]
Klaiber, A.C. and Levy, H.M., "An Architecture for Software-Controlled Data Prefetching," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 43-53

[KOOP93]
Koopman, P.J., "Perils of the PC Cache," *Embedded Systems Programming*, May 1993, pp. 26-34

[LAHA88]
Laha, S., Patel, J.H., and Iyer, R.K., "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Transactions on Computers*, Vol. 37, No. 11, November 1988, pp. 1325-1336

[LARU92]
Larus, J.R., "Rewriting Executable Files to Measure Program Behavior," *University of Wisconsin Computer Sciences Department Technical Report 083*, March 1992

[LARU93]
Larus, J.R., "Efficient Program Tracing," *IEEE Computer*, Vol. 26, No. 5, May 1993, pp.52-61

[LEBE94]
Lebeck, A.R. and Wood, D.A., "Cache Profiling and the SPEC Benchmarks: A Case Study," *IEEE Computer*, Vol. 27, No. 10, October 1994, pp. 15-26

[LIU93]
Liu, J.C. and Shahrier, S.M., "On Predictability and Optimization of Multiprogrammed Caches in Hard Real-Time Systems," *Technical Report 93-025*, Computer Science Department, Texas A&M University, 1993

[MART82]
Martz, H.F. and Waller, R.A., *Bayesian Reliability Analysis*, John Wiley & Sons, New York, 1982

[McCR91]
McCrackin, D.C. and Szabados, B., "Using Horizontal Prefetching to Circumvent the Jump Problem," *IEEE Transactions on Computers*, Vol. 40, No. 11, November 1991, pp. 1287-1291

[MILL95]
Milligan, M.K. and Cragon, H.G., "Processor Implementations Using Queues," *IEEE Micro*, Vol. 15, No. 4, August 1995, pp. 58-66

[MUEL94]
Mueller, F., "Static Cache Simulation and its Applications," Ph.D. Dissertation, Florida State University, August 1994

[NILS94]
Nilsen, K., "Problems with Caches for Multitasking Applications," *Computer Architecture News*, Vol. 22, No. 4, September 1994, pp. 79-80

[NOWI92]
Nowicki, G.J., "The Design and Implementation of a Read Prediction Buffer," Masters Thesis, Naval Postgraduate School, December 1992

[PEAR76]
Pearson, E.S. and Hartley, H.O., *Biometrika Tables for Statisticians, Volume 1*, Biometrika Trust, 1976

[POUR94]
Poursepanj, A., "The PowerPC Performance Modeling Methodology," *Communications of the ACM*, Vol. 37, No. 6, June 1994, pp. 47-55

[POWE93]
PowerPC 601 RISC Microprocessor User's Manual, Rev 1, Motorola, 1993

[SIMP89]
Simpson, D., "Real-Time RISCs," *Systems Integration*, July 1989, pp. 35-38

[SMIT78]
Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, Vol. 11, No. 12, December 1978, pp. 7-21

[SMIT82]
Smith, A.J., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 473-530

[SMIT85]
Smith, A.J., "Cache Evaluation and the Impact of Workload Choice," *12th Annual International Symposium on Computer Architecture Proceedings, Computer Architecture News*, Vol. 13, June 1985, pp. 64-73

[STAE93]
Staehli, R. and Walpole, J., "Constrained-Latency Storage Access," *IEEE Computer*, Vol. 26, No. 3, March 1993, pp. 44-53

[THIE87]
Thiebaut, D. and Stone, H.S., "Footprints in the Cache", *ACM Transactions on Computer Systems*, Vol. 5, No. 4, November 1987, pp. 305-329

[UHLI95]
Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., and Emer, J., "Instruction Fetching: Coping with Code Bloat," *Computer Architecture News*, Vol. 23, No. 2, May 1995, pp. 345-356

[VOLD81]
Voldman, J. and Hoevel, L. W., "The Software-Cache Connection," *IBM Journal of Research and Development*, Vol. 25, No. 6, November 1991, pp. 877-893

[WALK53]
Walker, H.M. and Lev, J., *Statistical Inference*, Holt, Rinehart and Winston, New York, 1953

[WOLF93]
Wolfe, A., "Software-Based Cache Partitioning for Real-Time Applications," *Proceedings of the Third Annual Workshop on Responsive Computer Systems*, September 1993

Vita

Michael Kevin Jack Milligan was born in Hamilton, Ontario, Canada on April 1, 1961, the son of Jack Edward Milligan and Clara Frances Milligan. After completing his work at the University of Detroit High School and Academy, Detroit, Michigan in 1979, he entered Michigan State University in East Lansing, Michigan. He received the degree of Bachelor of Science in Electrical Engineering in June 1983. Upon graduation, he entered the United States Air Force and was commissioned as a 2nd Lieutenant. Over a period of eight years, he worked on several research and development projects involving communications and computer systems. He received the Master of Business Administration from Western New England College in 1987 and the Master of Science in Electrical Engineering from the University of Massachusetts Lowell in 1989. In August 1991, he became an instructor of electrical engineering at the United States Air Force Academy in Colorado Springs, Colorado. In August 1993, he entered the Graduate School of the University of Texas at Austin.

Permanent Address: 17374 Parkside, Detroit, Michigan 48221

This dissertation was typed by the author using WordPerfect V 6.1.