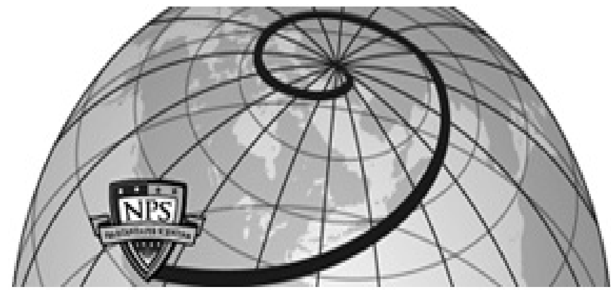
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations 1. Thesis and Dissertation Collection, all items

1998-06

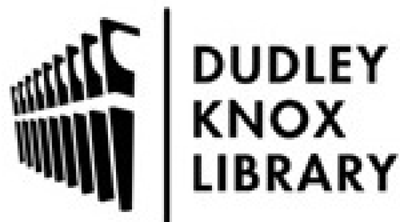
Architectural development and performance analysis of a primary data cache with read miss address prediction capability

Christensen, Kathryn S.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/32687

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

19980727 149

THESIS

ARCHITECTURAL DEVELOPMENT AND PERFORMANCE ANALYSIS OF A PRIMARY DATA CACHE WITH READ MISS ADDRESS PREDICTION CAPABILITY

by

Kathryn S. Christensen

June 1998

Thesis Co-Advisors: Douglas J. Fouts
Frederick Terman

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 1

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE ARCHITECTURAL DEVELOPMENT AND PERFORMANCE ANALYSIS OF A PRIMARY DATA CACHE WITH READ MISS ADDRESS PREDICTION CAPABILITY

5. FUNDING NUMBERS

6. AUTHOR(S) Kathryn S. Christensen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This work is part of an ongoing effort to bridge the cycle-time gap between high-speed processing units and lower-speed main memories through the use of memory hierarchies. Cache memory exploits the principle of locality by providing a small, fast memory between the processor and the main memory. The Predictive Read Cache (PRC) further improves the overall memory hierarchy performance by tracking the data read miss patterns of memory accesses, developing a prediction for the next access and prefetching the data into the faster cache memory. The PRC has been proven to significantly improve system performance when acting as a second-level cache. The purpose of this thesis is to simulate the effectiveness of the PRC as a first-level cache in the memory hierarchy using the same simulator developed to prove the effectiveness of the PRC as a second-level cache.

14. SUBJECT TERMS Predictive Read Cache; address prediction; memory bandwidth; memory latency; cache memory; memory systems

15. NUMBER OF PAGES 94

16. PRICE CODE

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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

ARCHITECTURAL DEVELOPMENT AND PERFORMANCE ANALYSIS OF A PRIMARY DATA CACHE WITH READ MISS ADDRESS PREDICTION CAPABILITY

Kathryn S. Christensen
Lieutenant, United States Navy
B.S., United States Naval Academy, 1992

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1998

Author: ______________________
Kathryn S. Christensen

Approved by: ______________________
Douglas J. Fouts, Thesis Co-Advisor

____
Frederick Terman, Thesis Co-Advisor

____
Herschel H. Loomis, Jr., Chairman
Department of Electrical and Computer Engineering

ABSTRACT

This work is part of an ongoing effort to bridge the cycle-time gap between high-speed processing units and lower-speed main memories through the use of memory hierarchies. Cache memory exploits the principle of locality by providing a small, fast memory between the processor and the main memory. The Predictive Read Cache (PRC) further improves the overall memory hierarchy performance by tracking the data read miss patterns of memory accesses, developing a prediction for the next access and prefetching the data into the faster cache memory. The PRC has been proven to significantly improve system performance when acting as a second-level cache. The purpose of this thesis is to simulate the effectiveness of the PRC as a first-level cache in the memory hierarchy using the same simulator developed to prove the effectiveness of the PRC as a second-level cache.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MEMORY HIERARCHY DESIGN 1

B. CACHE MEMORY 2

C. GOALS OF THE THESIS 6

D. THESIS OUTLINE 6

II. BACKGROUND OF THE PREDICTIVE READ CACHE 7

A. THE PREDICTIVE READ CACHE 7

B. THE INSTRUCTION PRC 8

C. THE CACHE AND PRC SIMULATOR 8

1. Address Traces 8

2. CaPSim 9

III. FIRST-LEVEL CACHE CONFIGURATION AND RESULTS 11

A. DEMAND-DRIVEN FIRST-LEVEL CACHE CAPSIM CONFIGURATION . 11

B. FIRST-LEVEL PRC CAPSIM CONFIGURATION 14

C. TRADITIONAL CACHE VS. IPRC SIMULATION RESULTS 20

1. Direct-Mapped First-level Cache Simulations 20

2. 4-Way Set-associative First-level Cache Simulations 23

3. Fully Associative First-level Cache Simulations 24

D. TRADITIONAL CACHE VS. PRC SIMULATION CONCLUSIONS 26

IV. THE DEVELOPMENT AND SIMULATION OF A DEMAND PRC 29

A. FIRST-LEVEL DEMAND PRC CAPSIM CONFIGURATION 29
B. FIRST-LEVEL DEMAND PRC SIMULATION RESULTS 30
1. Direct-Mapped First-level Cache Simulations 30
2. 4-Way Set-associative First-level Cache Simulations 33
3. Fully Associative First-level Cache Simulations 37
C. FIRST-LEVEL DEMAND PRC CONCLUSIONS 40

V. THE DEVELOPMENT AND SIMULATION OF A PRIORITY-DEMAND PRC 41
A. PRIORITY-DEMAND PRC CAPSIM CHANGES 41
B. FIRST-LEVEL PRIORITY-DEMAND PRC SIMULATION RESULTS 41
1. Direct-Mapped First-level Cache Simulations 41
2. 4-Way Set-associative First-level Cache Simulations 45
3. Fully Associative First-level Cache Simulations 48
C. FIRST-LEVEL PRIORITY-DEMAND PRC CONCLUSIONS 51

VI. CONCLUSIONS 53
A. EFFECTIVENESS OF THE PRC AS A FIRST-LEVEL CACHE 53
B. SUGGESTION FOR FUTURE DEVELOPMENT 53

APPENDIX A. AN EXAMPLE CAPSIM CONFIGURATION FILE 55

APPENDIX B. AN EXAMPLE CAPSIM CONFIGURATION FILE 57

APPENDIX C. AN EXAMPLE CAPSIM LOG FILE 59

APPENDIX D. AN EXAMPLE OUTPUT FILE FOR THE CPU MODULE 63

APPENDIX E. AN EXAMPLE OUTPUT FILE FOR THE CACHE MODULE 65

APPENDIX F. AN EXAMPLE OUTPUT FILE FOR THE PRC MODULE 69

APPENDIX G. AN EXAMPLE OUTPUT FILE FOR THE BUFFER MODULE 73

APPENDIX H. AN EXAMPLE OUTPUT FILE FOR THE MAIN MEMORY MODULE 75

LIST OF REFERENCES 77

INITIAL DISTRIBUTION LIST 79

LIST OF FIGURES

Figure 1. Memory Hierarcy 2

Figure 2. First-level Cache-Only Memory Hierarchy 11

Figure 3. First-level PRC Configuration 14

Figure 4. Original Memory Hierarchy Handshaking 17

Figure 5. First-level PRC Handshaking 18

Figure 6. Revised First-level PRC Handshaking 19

Figure 7. Hit Rate vs. Cache Size for Direct-mapped cache, Kenbus20 21

Figure 8. Hit Rate vs. Cache Size for Direct-mapped Cache, Kenbus80 21

Figure 9. Access Time vs. Cache Size for Direct-mapped cache, Kenbus20 22

Figure 10. Access Time vs. Cache Size for Direct-mapped cache, Kenbus80 22

Figure 11. Hit Rate vs. Cache Size for 4-way Set-associative cache, Kenbus20 23

Figure 12. Hit Rate vs. Cache Size for 4-way Set-associative cache, Kenbus80 23

Figure 13. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus20 24

Figure 14. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus80 24

Figure 15. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus20 25

Figure 16. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus80 25

Figure 17. Access Time vs. Cache Size for Fully Associative Cache, Kenbus20 26

Figure 18. Access Time vs. Cache Size for Fully Associative Cache, Kenbus80 26

Figure 19. Hit Rate vs. Cache Size for Direct-mapped cache, Kenbus20 30

Figure 20. Hit Rate vs. Cache Size for Direct-mapped Cache, Kenbus80 31

Figure 21. Access Time vs. Cache Size for Direct-mapped Cache, Kenbus20 31

Figure 22. Read Access Time vs. Cache Size for Direct-mapped Cache, Kenbus80 32

Figure 23. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus20 32

Figure 24. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus80 33

Figure 25. Hit Rate vs. Cache Size for 4-Way Set-associative Cache, Kenbus20 34

Figure 26. Hit Rate vs. Cache Size for 4-way Set-associative Cache, Kenbus80 34

Figure 27. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus20 35

Figure 28. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus80 35
Figure 29. Speed up vs. Cache Size for 4-way Set-associative Cache, Kenbus20 36
Figure 30. Speed Up vs. Cache Size for 4-Way Set Associative Cache, Kenbus80............. 36
Figure 31. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus20........................ 37
Figure 32. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus80........................ 37
Figure 33. Access Time vs. Cache Size for Fully Associative Cache, Kenbus20 38
Figure 34. Access Time vs. Cache Size for Fully Associative Cache, Kenbus80 38
Figure 35. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus20...................... 39
Figure 36. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus80...................... 39
Figure 37. Hit Rate vs Cache Size for Direct-mapped Cache, Kenbus20............................. 42
Figure 38. Hit Rate vs. Cache Size for Direct-mapped Cache, Kenbus80............................ 42
Figure 39. Access Time vs. Cache Size for Direct-mapped Cache, Kenbus20 43
Figure 40. Access Time vs. Cache Size for Direct-mapped Cache, Kenbus80 43
Figure 41. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus20.......................... 44
Figure 42. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus80.......................... 44
Figure 43. Hit Rate vs. Cache Size for 4-way Set-associative Cache, Kenbus20................. 45
Figure 44. Hit Rate vs. Cache Size for 4-way Set-associative Cache, Kenbus80................. 45
Figure 45. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus20 46
Figure 46. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus80 46
Figure 47. Speed Up vs. Cache Size for 4-Way Set-associative Cache, Kenbus20.............. 47
Figure 48. Speed Up vs. Cache Size for 4-Way Set-associative Cache, Kenbus80.............. 47
Figure 49. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus20........................ 48
Figure 50. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus80........................ 48
Figure 51. Access Time vs. Cache Size for Fully Associative Cache, Kenbus20 49
Figure 52. Access Time vs. Cache Size for Fully Associative Cache, Kenbus80 49
Figure 53. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus20...................... 50
Figure 54. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus80...................... 50

LIST OF TABLES

Table 1. Traditional Cache Configuration 12
Table 2. Buffer Module Configuration 13
Table 3. Main Memory Configuration 13
Table 4. First-level PRC Configuration 14
Table 5. Transaction Request Format 15

I. INTRODUCTION

A. MEMORY HIERARCHY DESIGN

> Ideally one would desire an indefinitely large memory capacity such that any particular...word would be immediately available.... We are...forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A.W. Burks, J.H. Goldstine, and J. von Neumann, Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946) [Ref. 1:p. 372]

The early computer designers recognized the need for memory hierarchy to diminish the cycle-time gap between processors and data storage devices. A von Neumann machine executes a program in the following manner: the CPU repeatedly fetches the instruction from memory as well as any operands the instruction requires, it performs the indicated operation and then, frequently, writes the result back to memory. These recurrent memory accesses have become the limiting factor in overall system performance.

Processor cycle time has dramatically decreased over the years while memory technology has fallen behind. In particular, Very Large Scale Integrated (VLSI) technology enables processors to complete the computation portion of the instruction cycle much faster, making the memory access times even more of a system performance issue.

This problem leads to a trade off between size, speed and cost of the main memory. One solution is to design the main memory with the same technology used for the CPU. This would be technically impractical and prohibitively expensive to implement on such a large scale. Instead, the concept of memory hierarchies was developed as a more cost-effective solution to this problem.

The design of a memory hierarchy consists of placing smaller, faster, more expensive memories between the processor and the larger, less expensive, slower memory. These memories have been named cache memories. Figure 1 illustrates a general case of a

memory hierarchy. The cache memory level of the hierarchy can be multiple levels of caches, consisting of a first-level cache (the cache closest to the CPU), second-level cache, etc. The terms on-chip cache and off-chip cache refer to the physical location of the cache: either on the same chip as the processor or outside the chip.

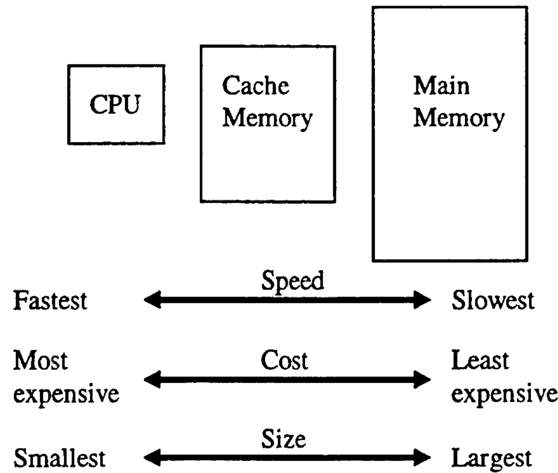


Figure 1. Memory Hierarcy

B. CACHE MEMORY

The concept of cache memory operation is based on the principle of locality. There are two types of locality: spatial and temporal. Spatial locality refers to the concept that when a memory address is referenced, the memory addresses near the one referenced are likely to be referenced in the near future. Temporal locality is the concept that when a memory address is referenced, it is likely to be referenced again in the near future [Ref. 2:p. 344]. Cache memory exploits these principles of locality by storing copies of the recently accessed main memory data and instructions in the cache. Temporal locality predicts that the same reference will be used again soon and the next time the data will be fetched from the faster cache memory instead of the slower main memory. The cache is updated with a block which is larger in size than the word requests of the CPU. Once the data is fetched from main memory in the original request, it is stored in the cache memory along with a few word addresses surrounding it. Therefore, if the CPU requests an address near the original

one (spatial locality) the data will be found in the faster cache memory instead of needing to be fetched from main memory.

One method of measuring performance of cache memory is to measure the cache hits and cache misses. A cache hit occurs when the CPU finds the requested memory address in the cache, a cache miss occurs when the requested memory address is not located in the cache. Cache hits and misses are further divided into the categories of read hits, read misses, write hits and write misses. The cache hit ratio is simply the number of cache hits divided by the number of requests. The miss ratio is the number of CPU requests that miss in the cache divided by the total number of requests [Ref. 1:p. 43]. Cache hit ratios are not enough to accurately evaluate system performance. Przybylski [Ref. 3:p. 5] warns of the dangers of focusing on the "time-independent" statistics. To improve system performance, the entire system must be optimized, not merely a single aspect.

There are three different types of misses that may occur in a cache: compulsory, capacity and conflict. A compulsory miss is one which could not be avoided, often the first access to a data address [Ref. 4:p. 245]. A capacity miss occurs when the cache is not large enough to hold all of the blocks required during program execution. In this case, a request is made to the cache which requires a block which was once replaced to be retrieved again from memory [Ref. 1:p. 390]. A conflict miss occurs through a request to a direct-mapped or set-associative cache when too many requested blocks map to the same set [Ref. 1:p.390].

Overall system performance is dependant on the miss penalty as well as the hit/miss ratios. The miss penalty is defined as the time (in clock cycles) it takes the CPU to fetch the required data from main memory upon a cache miss. Specifically:

$$\text{Miss penalty} = \text{Memory access time} / \text{Clock period}$$

Speedup is a performance measure which compares the relative performance between two configurations. Specifically, in this thesis, speedup is defined as:

$$\text{Speedup} = (\text{Read Access Time}_{cache} - \text{Read AccessTime}_{PRC}) / \text{Read AccessTime}_{cache}$$

Cache performance is effected by many different parameters: cache size, block size, associativity, replacement policy, write policy, and write-miss policy. Cache size refers to the number of bytes the cache can store. Block size is the fixed size of memory which is transferred to the cache at a time. Associativity is the mapping function between the cache memory and the main memory and is necessary because the cache memory is smaller than the main memory.

There are three main types of cache associativity: direct-mapped, fully associative and set-associative. In a direct-mapped cache, each main memory location can only be mapped into a specific cache location. If there is already data occupying that location, then that data must be removed from the cache. In a fully associative cache, any main memory location can be mapped into any cache location. In the fully associative case, data needs to be removed from the cache only if the entire cache is full. Set-associative is in between direct-mapped and fully associative. The set-associative cache maps a certain portion of main memory to a designated portion of the cache memory, called a set. Data is replaced in the cache only when the set to which the incoming data is mapped is full. The set a block is mapped to is determined by:

(*block address*) MOD (*number of blocks in cache*) [Ref. 1. p. 376].

Block address is defined as the actual main memory address divided by the block size in bytes. The cache is said to be n-way set-associative, where n is the number of blocks in a set. n is calculated by:

(*number of blocks in cache*)/(*number of sets in cache*)

or

(*cache size in bytes*) /[(*block size in bytes*) *(*number of sets in cache*)]

Direct-mapped is actually a special case of set-associative with an associativity of one. Fully associative is also a special case of set-associative where n is equal to the number of blocks in the cache.

When there is no room in the cache for the incoming block, the cache uses a replacement policy to choose which block to remove to make room. No replacement policy is needed in a direct-mapped cache since there is only one place in the cache a given

memory address can be mapped. Therefore, if it is being used, the data in that location must be removed. The most common replacement policies are: Least Recently Used (LRU), First In First Out (FIFO) and random. LRU tracks the usage statistics on each block in the set and chooses the one for replacement which is the oldest. FIFO designates the oldest block in the cache for replacement. Random replacement chooses the candidate for replacement at random from all of the blocks in the set.

There are two major types of write policies: write back and write through. In a write-through cache, data is written to the cache at the same time it is written to the main memory. This policy slows down the overall system speed because the speed of all writes is limited by the main memory write speed. There are two advantages of a write-through cache: the hardware is less complex and the cache is always coherent with the data in main memory. Write back only updates the cache memory upon a write, main memory does not get updated until that block is chosen for replacement.

The cache write-miss policy determines the sequence of events which occur when a CPU write request misses in the cache. Common types of write-miss policies are: write allocate and write around. The write allocate policy loads the block into the cache and then modifies the data according to the write policy in effect. In a write around cache the CPU writes to the block in main memory, completely bypassing the cache. The block is not loaded into the cache on a write miss when a write around policy is in effect.

Cache memory is sometimes divided into a hierarchy within itself. The cache memory closest to the CPU is called the level one or L1 cache. The level denoted by the largest number is the cache which is located closest to the main memory. It is also common for there to be separate caches for instructions and for data, called a split level cache. Instructions and data have different reference patterns and splitting them apart allows separate cache designs for data and instruction caches. Split level caches further increase the performance by doubling the cache bandwidth.

The large number of parameters which determine the performance of cache memory has launched a whole field of study in cache design. Performance optimization is extremely difficult due to the large number of factors involved. New technological advances and the

complexity surrounding cache design indicate that the study of cache design will continue to be an intense area of research efforts.

C. GOALS OF THE THESIS

The goal of this thesis is to simulate and evaluate the performance of the Predictive Read Cache as a first-level data cache in a memory hierarchy with only a level one cache. The Cache and PRC simulator (CaPSim) [Ref. 5] will be used for this evaluation.

D. THESIS OUTLINE

The remainder of this thesis is organized as follows. Chapter II discusses the background of the PRC research. The fundamentals of both the Instruction Predictive Read Cache (iPRC) and the Data Predictive Read Cache (dPRC) algorithms will be described. Hardware architectures are presented and read/write operations are discussed. The trace driven simulator and the address traces used in the simulations will be presented. Chapter III discusses the reconfigurations needed to CaPSim to accurately simulate a memory hierarchy with only a single level cache and the changes needed to simulate the PRC as a first-level cache. The results of these simulations will be presented. A new algorithm is presented in Chapter IV: a demand Predictive Read Cache. Simulations are described and compared with a purely demand-driven cache. Chapter V presents an improved version of the demand PRC which was developed to reduce the average read access time of the demand PRC in Chapter IV. Finally, Chapter VI contains conclusions and suggestions for future work.

II. BACKGROUND OF THE PREDICTIVE READ CACHE

A. THE PREDICTIVE READ CACHE

The Predictive Read Cache (PRC) is a special cache designed by Fouts and Billingsley [Ref. 6]. It was originally intended to be implemented as a second-level data cache. The PRC uses a prediction algorithm to predict the data address of the next primary data cache miss. The data at the predicted address is then prefetched into the PRC, awaiting the primary cache's request.

The PRC's prediction algorithm is based upon the fact that most data requests are to sequential data structures stored in memory. The PRC predicts the next primary cache miss by simply taking the difference of the last two data read address requests from the primary cache and adding that difference to the last data miss address. The PRC then makes a request to memory to prefetch the data at the predicted address.

For example, the CPU makes a request for data at the address of 10001000. This request misses in the primary data cache, which forwards this request to both the main memory and the PRC. The PRC cannot make a prediction at this point since it is the first request. The next request is for data at address 10001004. Again, this misses in the primary data cache and is forwarded to both main memory and the PRC. This time the PRC makes a prediction based on the following simple calculation: 10001004 + (10001004-10001000) = 10001008. The PRC will then prefetch the data from address 10001008 from main memory and store it in the PRC. Assuming that the CPU is accessing a data array with each element consisting of 4 bytes, the next request should be a read hit in the PRC, thus preventing the long cycle time required to fetch it from main memory.

The PRC requires additional storage for the most recent miss address (MRMA) and the previous miss address (PRMA) for each cache block. The PRC algorithm also requires the addition of a subtracter-adder pair (or just a subtracter with a 1-bit offset for the

MRMA) to calculate the displacement between the data read miss addresses. The PRC demonstrated a significant improvement in performance over a second-level cache [Ref 7].

B. THE INSTRUCTION PRC

The Instruction PRC (iPRC) algorithm was designed by Altmisdort and fully described in reference 5. The goal of the iPRC is to improve performance during program branches and context switches by reducing the miss penalty on compulsory misses. The iPRC does this by maintaining a relationship between the addresses of the read misses and the addresses of the instructions that cause the read misses [Ref. 5, p. 9].

The iPRC uses a similar architecture to the original PRC and adds additional storage for the instruction tag for each block. It also requires that an instruction bus be added between the CPU and the iPRC (transparent to the first-level cache) to provide the instruction addresses of the data requests.

The iPRC operates in a similar manner to the original PRC: when two read misses occur, a signed displacement is determined between the MRMA and the PRMA. This displacement is added to the MRMA to predict the address of the next read miss.

The iPRC performance was simulated using address trace simulations and the results were documented in reference 5. The iPRC provides a significant improvement in performance over a second-level cache and a nominal performance increase over the dPRC algorithm.

C. THE CACHE AND PRC SIMULATOR

The Cache and PRC Simulator (CaPSim) is an address-trace driven simulator developed by Altmisdort to simulate a memory hierarchy which can be configured for either traditional, original dPRC or iPRC caches of multiple levels [Ref. 5].

1. Address Traces

CaPSim uses address traces collected from the SPEC SDM (System Development Multitasking) benchmark programs on the SPARC platform. These address traces were

collected by the BYU BACH system [Ref. 8]. The benchmarks used for the simulations were the Kenbus20 and the Kenbus80 benchmark programs. Kenbus20 models the behavior of a Unix operating system in a multitasking, educational environment. Kenbus20 simulates the demands made by twenty users on the system at one time. Kenbus80 models the same multitasking environment but with eighty users on the system. The Kenbus80 benchmark has more context switching and thus more compulsory misses than does the Kenbus20 benchmark. These traces were chosen because they represent the most demanding environment for a predictive cache with context changes occurring frequently due to the heavy multitasking load.

There are two types of address traces: the original BYU format address trace and the PRC format for use with the iPRC cache. The PRC format includes the necessary instruction tag information to make the proper predictions. Reference 5 describes at length the use of the address traces and the software conversion tool.

2. CaPSim

The Cache and PRC Simulator (CaPSim) is written in C++ code using object-oriented programming techniques. CaPSim may be configured to simulate different memory configurations.

The CaPSim architecture is centered around the concept of a generic memory module. Up to five different types of memory modules can currently be defined from the generic: CPU, Cache, PRC, Buffer, and main memory. CaPSim has been programmed so that new memory modules, such as disk drives or a virtual memory system, may be added to the memory hierarchy by simply making small changes to the CPU class and programming a new module with adherence to the generic memory module format [Ref 2: p.70].

CaPSim comes complete with an integrated, interactive debugger. The debugger displays the inter-cycle events as well as the request-respond handshaking of the modules. Its operation and capabilities are described fully in reference 5.

III. FIRST-LEVEL CACHE CONFIGURATION AND RESULTS

A. DEMAND-DRIVEN FIRST-LEVEL CACHE CAPSIM CONFIGURATION

Some minor changes were necessary to allow CaPSim to simulate the configuration shown in Figure 2.

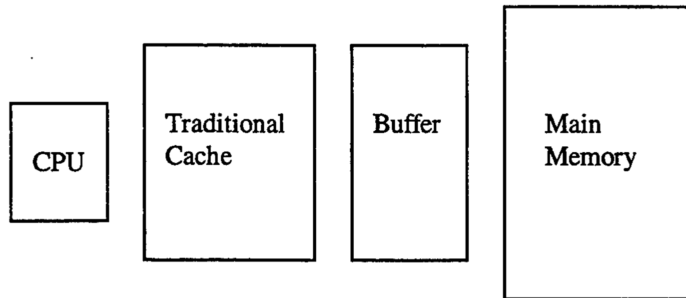


Figure 2. First-level Cache-Only Memory Hierarchy

First-level caches of sizes varying from 256 Bytes to 512 Kbytes were simulated. All sizes were simulated for three different degrees of associativity: direct-mapped, fully associative and four way set-associative. Table 1 delineates the remaining properties which were constant throughout the simulations.

Block size and sub block size were 16 and 8 bytes respectively. The sub block size is the smallest size which maintains an independent valid bit. The fetch size determines the size of the memory request made after a read miss in the cache. The specification of a fetch size allows the cache to fetch multiple blocks from the cache upon a single read miss. In this configuration, a single-block fetch is simulated. The transfer size determines the bus width between the cache and the CPU.

The write policy is write through and the write-miss policy is write around. Both of these policies were described in Chapter I. The wrapping-fetch policy determines the direction of fetches from higher memory levels during a block update [Ref. 5:p. 90].

The access time determines the number of cycles expended to access the cache for either a read or a write request. The read/write hit and miss times are penalties imposed in

addition to the access time to model an excessive delay imposed by the architecture, in this case they are all set to zero.

The cache block buffer is enabled because in the case of the PRC (with which these simulations results will later be compared) the block buffer is always enabled. When the Read Forwarding policy is in effect, the missed word is fetched from main memory first and then the word is forwarded to the CPU at the same time it is written to the block buffer. This policy allows the cache to continue servicing CPU requests while the rest of the block is being updated in the cache [Ref. 4:p.83]. The Read Forwarding option is not used with the Cache Module because it is not an option with the CaPSim PRC module.

Parameter Name	Parameter Value	Parameter Name	Parameter Value
Block Size	16 bytes	Access Time	1 cycle
Sub-block Size	4 bytes	Write Hit Time	0
Fetch Size	16 bytes	Write Miss Time	0
Transfer Size	4 bytes	Read Hit Time	0
Replacement Policy	LRU	Read Miss Time	0
Write Policy	Write Through	Block Buffer Transfer Time	1 cycle
Write Miss Policy	Write Around	Enable Block Buffer	Yes
Wrapping Policy	Wrap Up	Search Block Buffer	Yes
		Read Forward	No

Table 1. Traditional Cache Configuration

The buffer module contains both a read and a write buffer. The buffers compensate for the difference in data flow rate during transfers between the cache and main memory. For instance, the write buffer allows the processor to continue execution as soon as the data is written into the buffer, instead of waiting for the slower main memory to complete the write.

The buffer parameters are constant throughout all simulations and are shown in Table 2. The read and write buffer sizes are eight and four bytes respectively. The write buffer block size refers to the number of bytes which can be stored in a single buffer line. This allows the buffer to combine adjacent write requests into a single request. Enforce priorities ensures that the highest priority requests are serviced first in the buffer. The

"remove read and write duplicates" parameters allow the buffer to combine duplicate requests into a single request. Search Read Buffer parameter allows the buffer to update the data in the read buffer from the write buffer in the case of a buffer write hit. The Search Write Buffer parameter allows the buffer module to conduct a search to determine if a read request will hit in the write buffer.

Parameter	**Value**
Read Buffer Size	8 bytes
Write Buffer Size	4 bytes
Write Buffer Block Size	16 bytes
Enforce Priorities	Yes
Remove Read Duplicates	Yes
Remove Write Duplicates	Yes
Search Read Buffer	Yes
Search Write Buffer	Yes

Table 2. Buffer Module Configuration

Table 3 shows the main memory module parameters used for all simulations. Access time refers to the number of cycles required for main memory to access the first word of a transfer. The remaining words are accessed at the "transfer time" rate of one per cycle. The transfer size determines the bus width between the main memory module and the buffer.

Parameter	Value
Access Time	5 cycles
Transfer Time	1 cycle
Transfer Size	4 cycles

Table 3. Main Memory Configuration

In order to successfully complete the baseline demand-driven cache simulations, there was a minor change which was made to the CaPSim program itself. Specifically, an error occurred when the cache was designated as a write-through cache and the incoming write request to the cache was registered as a pending request because the cache was busy. The request was not getting propagated to the buffer at any time in the CaPSim code. This

caused the CPU to wait indefinitely for the response to its write request. This was fixed by adding the proper code to propagate the request to the subordinate modules.

B. FIRST-LEVEL PRC CAPSIM CONFIGURATION

The memory hierarchy is similar to the hierarchy used in the simulations in part A, except the traditional cache is replaced with a PRC. The configuration is shown in Figure 3.

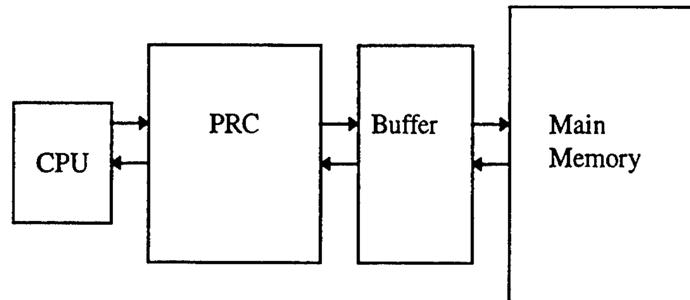


Figure 3. First-level PRC Configuration

The configuration of the main memory and buffer modules remains the same as they did for the simulations in Part A. The configuration of the PRC is shown in Table 4.

Parameter	Value
Block Size	16 bytes
Sub-Block Size	4 bytes
Fetch Size	16 bytes
Transfer Size	4 bytes
Replacement Policy	LRU
Write Policy	Write Through
Access Time	1 cycle
Read Hit Time	0
Read Miss Time	0
Write Hit Time	0
Write Miss Time	0
Block Buffer Transfer	1 cycle

Table 4. First-level PRC Configuration

The parameters are almost identical to those used in Section A of this chapter. The write-miss policy is not specified since CaPSim is programmed to always treat the PRC as a write around cache. The block buffer is not specifically enabled since CaPSim always enables the PRC block buffer and the searching of the block buffer. CaPSim does not offer the read forward option for the PRC so it is not a valid parameter to specify.

Many aspects of the CaPSim program itself had to be modified to allow the simulation of a PRC first-level cache. CaPSim was written with the main purpose of simulating the PRC as a second-level cache with a traditional first-level cache. Although it has the flexibility to assume other configurations, most of the other configurations had not been fully tested and many modifications to the C++ code were necessary.

The first reconfiguration needed was in the inter-module handshaking. Handshaking is the means of communication between the modules. The handshaking requests are used by the modules to make write or read requests from each other and to respond when the requests are completed. Table 5 below shows the memory request format.

Field	Size
Source ID	unsigned integer
Match ID	unsigned integer
Priority	integer
Total Size	integer
Data Address	AddressType
Instruction Address	AddressType
Transaction Type	{Read, Write, Cancel}
Minimum Size	integer
Drop Counter	integer
Original Address	AddressType
Original Size	integer
Victim Block	integer

Table 5. Transaction Request Format

The Source ID field designates where the request is originating from and therefore, where the response must be returned. The match ID is used when two modules are sharing a request. It ensures that both modules receive the proper response. The Data Address field holds the data address of the request and the Instruction Address field holds the instruction address. The Priority field specifies the priority of the request. The Total Size indicates the size of the current request. The Transaction Type indicates the type of transaction requested. Originally, the choices were Read, Write and Cancel. The Minimum Size field is used to determine if the minimum size of the transfer has occurred to see if the transaction may be interrupted or not. The Drop Counter is used by the Buffer Module to specify the number of tries a transaction is allowed before it is dropped out of the buffer. When used, the counter is decrements by one every time a transaction is canceled due to a higher priority transaction. Original Size and Original Address are used by the buffer module to restore the original parameters after the transaction had been modified by the module. The Victim Block field holds the place in that cache that this data is to replace.

Typically, the requests are made by a higher-level memory module to a lower-level module. The higher-level module changes the Source ID field to its own ID, therefore ensuring that the response is sent through that module on its way back to the CPU. Since the PRC was originally designed to be a second-level cache, the CaPSim PRC module is not programmed to handle request and response handshaking in the same way as the Cache Module, which is assumed to be the primary data cache in the hierarchy.

In a memory hierarchy with a traditional first-level cache and a PRC second-level cache, write-miss requests are handled in such a way that the PRC does not receive the response. Upon a write miss, the primary cache will send a request to the PRC and the PRC is programmed to immediately forward the request to the buffer, without changing the Source ID field of the request to its own Source ID. Leaving the Source ID field set to the primary cache module ID results in the primary cache directly receiving the responses to write-miss requests, completely bypassing the PRC module (Figure 4).

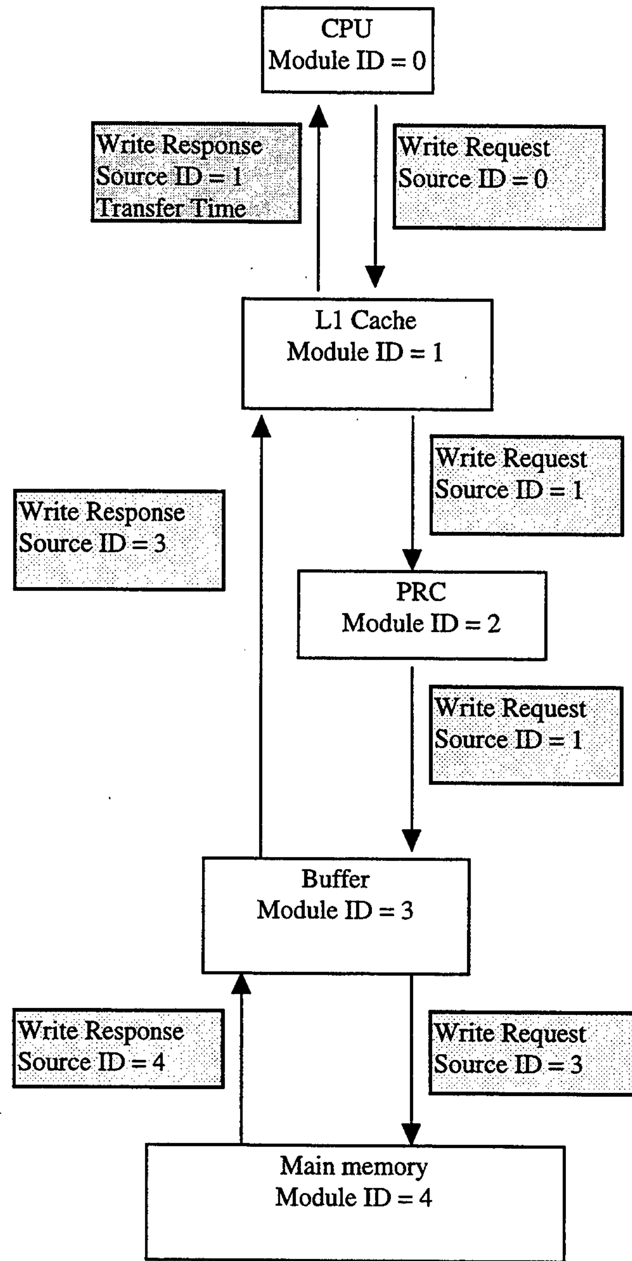


Figure 4. Original Memory Hierarchy Handshaking

This works very well in the memory configuration with a traditional-type first-level cache, which is the memory configuration used by Altmisdort [Ref. 5]. The CaPSim cache module is programmed to receive the response from the buffer and then calculate the appropriate transfer time, which is then forwarded to the CPU. Once the write response is

received by the CPU, it waits the appropriate time until the transfer is complete and then the CPU transitions out of the write stall state to fetch the next instruction.

The PRC Module behaves in the same manner when it is the primary cache as when it is the secondary cache. As in the previous case, the PRC receives the write request from the CPU and it forwards the request to the Buffer Module without changing the Source ID to its own Module ID. The buffer then responds directly to the CPU. In this way, the correct transfer time is not calculated when the buffer responds to the CPU (since that is programmed into the Cache Module) (Figure 5). This becomes a problem when the CPU prematurely transitions out of the write stall state and begins executing the next instruction before the write transfer is complete.

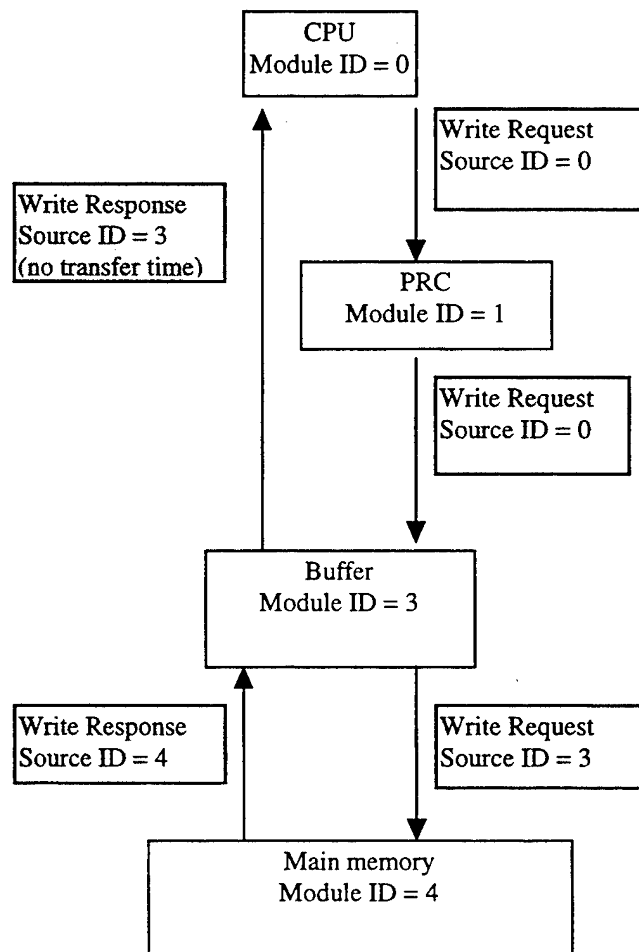


Figure 5. First-level PRC Handshaking

The solution involved modifying the PRC module so that it could handle write requests and responses as a first-level cache. The new PRC module includes the ability to modify the Source ID of write requests to its own Module ID. It further includes the ability to receive write responses, calculate the transfer time and propagate the response to the CPU (Figure 6).

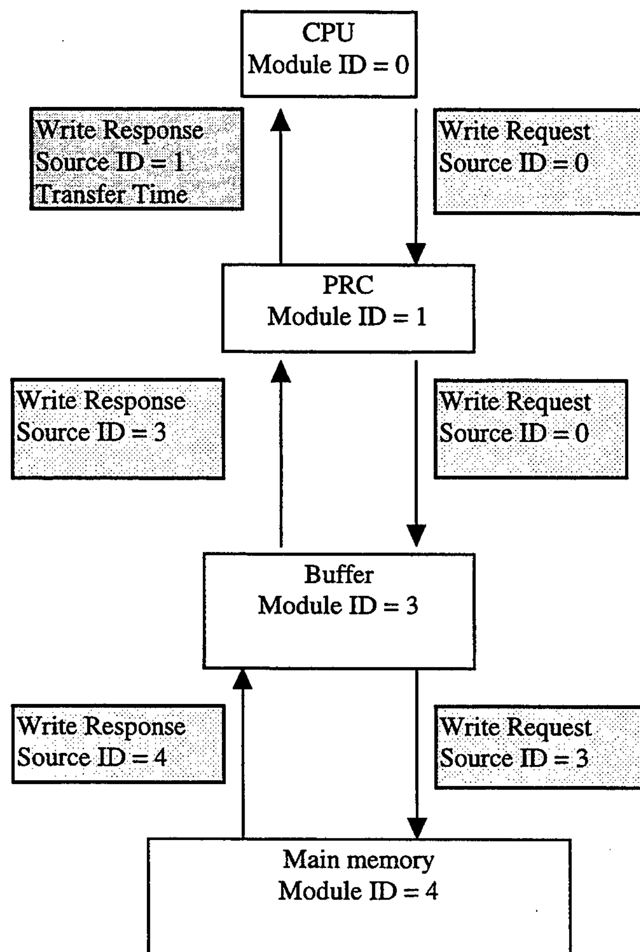


Figure 6. Revised First-level PRC Handshaking

A similar problem existed with the read request handshaking sequence. As with the write request, the PRC Module was programmed to maintain the original Source ID of the request and propagate it to its slave module. Also, the only type of read response the PRC module was programmed to receive was prefetch requests. To distinguish between a CPU-

generated read request and a PRC-generated prefetch request, a new type of request had to be created and included in the type definition of "transaction type". The PRC-generated prefetch requests are designated a transaction type named "Prefetch". The CPU-generated requests are a transaction type named "Read". The new PRC module will update the source ID of a CPU-generated read request to its own module ID. This ensures the response will be sent through the PRC. Upon receipt of a response, the PRC is able to distinguish between a prefetch response and a read response. In the case of a read response, the PRC will propagate the response to the CPU and, in the case of a response to a prefetch request, the PRC will not propagate the response to the CPU.

The next problems encountered were with the number of cancels occurring in the buffer module. With a PRC as a first-level cache, nearly every request made by the CPU, resulted in the PRC sending a prefetch request. This caused the buffer module to fill quickly and the need to cancel transactions happened more frequently. Problems arose when a request from the CPU was canceled and the CPU would remain in a stalled state forever because it did not receive an appropriate response. Assigning the prefetch requests a lower priority than the CPU requests ensured the prefetch requests would be canceled before the more important CPU requests.

C. TRADITIONAL CACHE VS. IPRC SIMULATION RESULTS

Figures 7-18 show the simulation results for direct-mapped cache, four-way set-associative cache and fully associative Demand Driven Cache(DDC) and Predictive Read Cache(PRC), respectively. Read hit rate and read access time are indicated. Results are displayed for both the Kenbus20 and the Kenbus80 benchmarks.

1. Direct-Mapped First-level Cache Simulations

The direct-mapped first-level cache simulations are conducted with the traditional demand driven cache and the PRC as first-level caches. The first-level cache size is varied between 256 bytes and 512 Kbytes, with each simulation increasing the size by a factor of

two. Figures 7-10 summarize the results for the read hit rate and average read access times respectively.

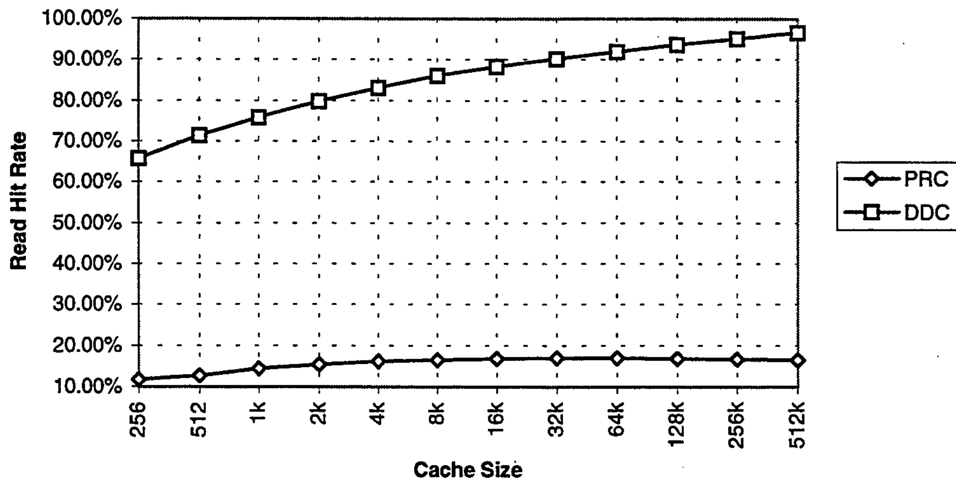


Figure 7. Hit Rate vs. Cache Size for Direct-mapped cache, Kenbus20

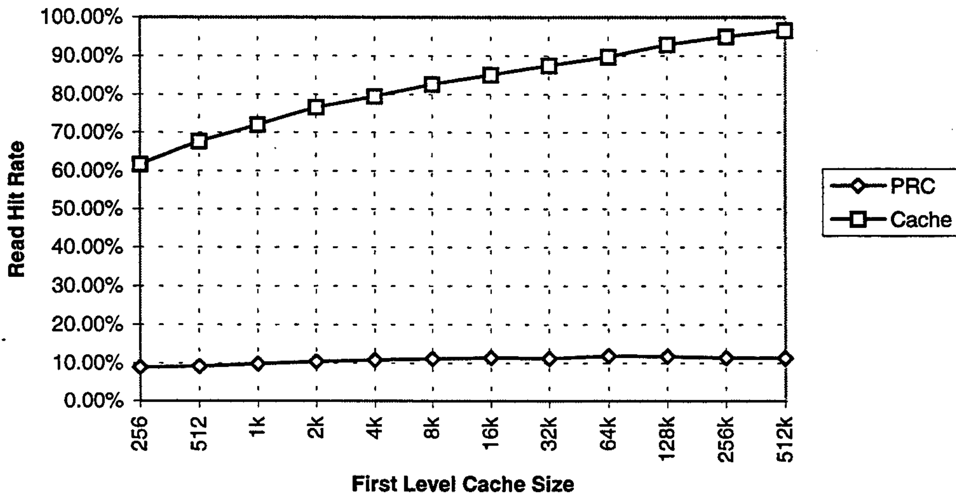


Figure 8. Hit Rate vs. Cache Size for Direct-mapped Cache, Kenbus80

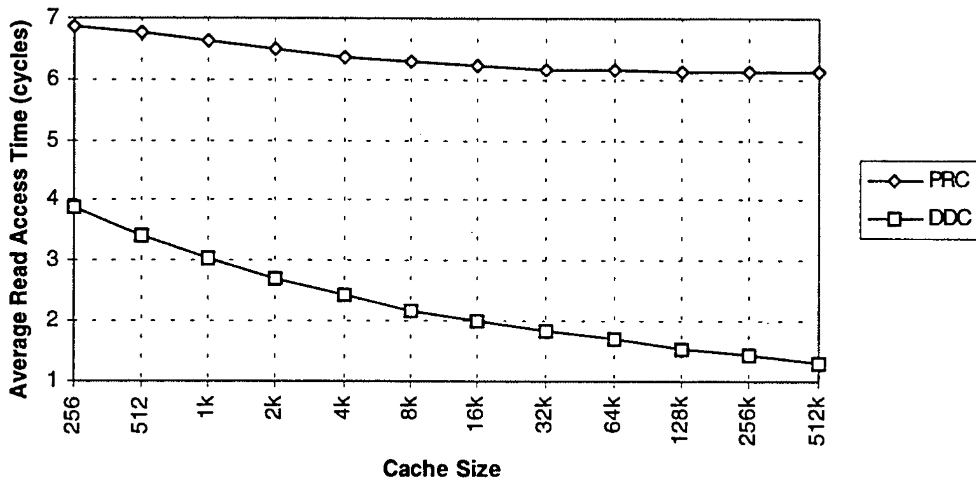


Figure 9. Access Time vs. Cache Size for Direct-mapped cache, Kenbus20

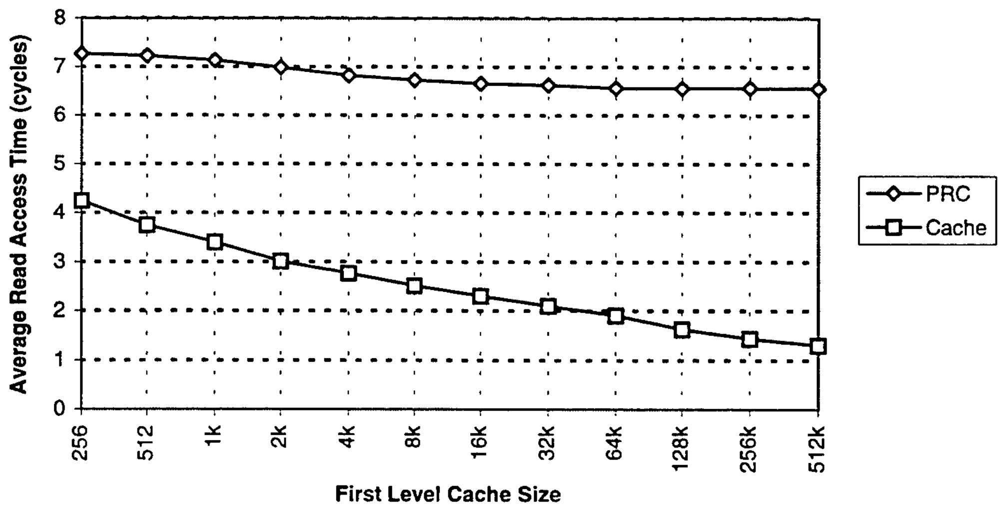


Figure 10. Access Time vs. Cache Size for Direct-mapped cache, Kenbus80

2. 4-Way Set-associative First-level Cache Simulations

The first-level cache simulations were repeated with the same cache sizes but with 4-way set associativity. The results are summarized in Figures 11-14.

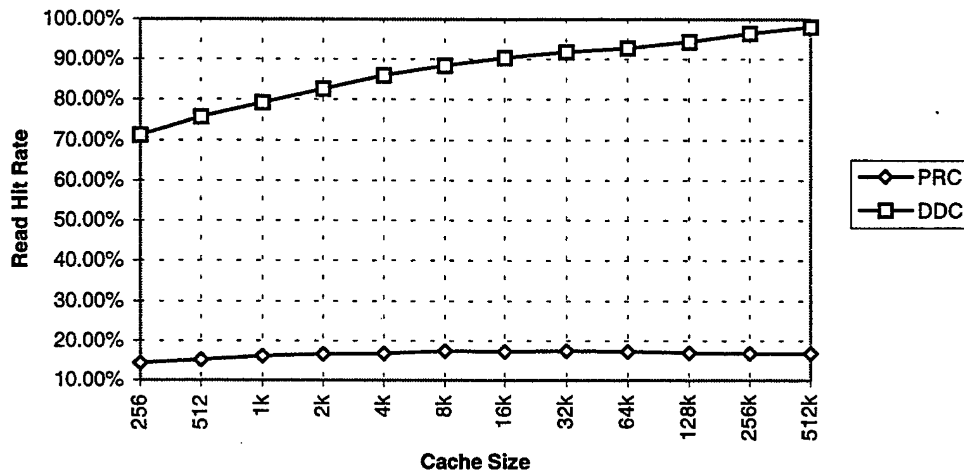


Figure 11. Hit Rate vs. Cache Size for 4-way Set-associative cache, Kenbus20

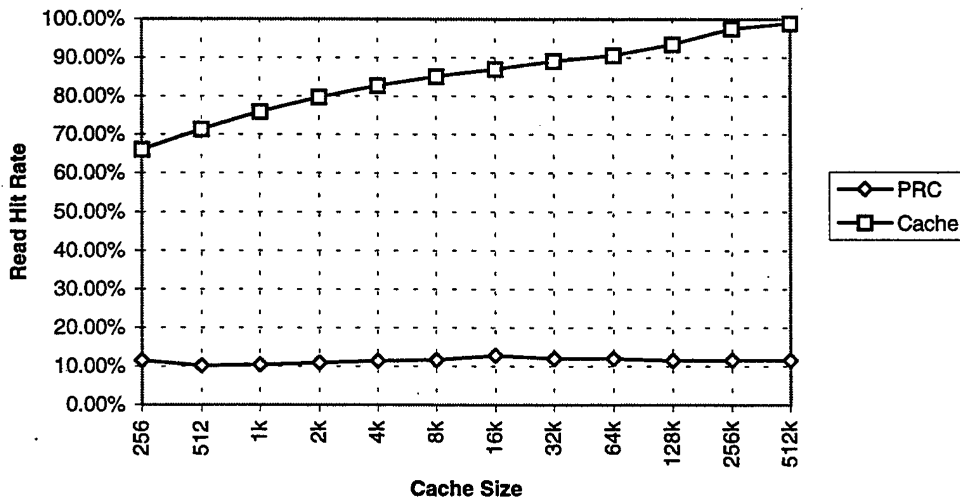


Figure 12. Hit Rate vs. Cache Size for 4-way Set-associative cache, Kenbus80

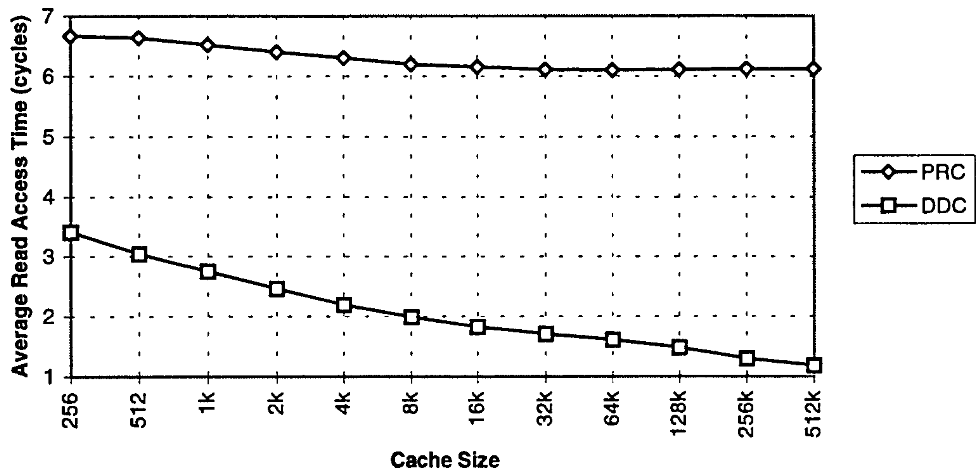


Figure 13. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus20

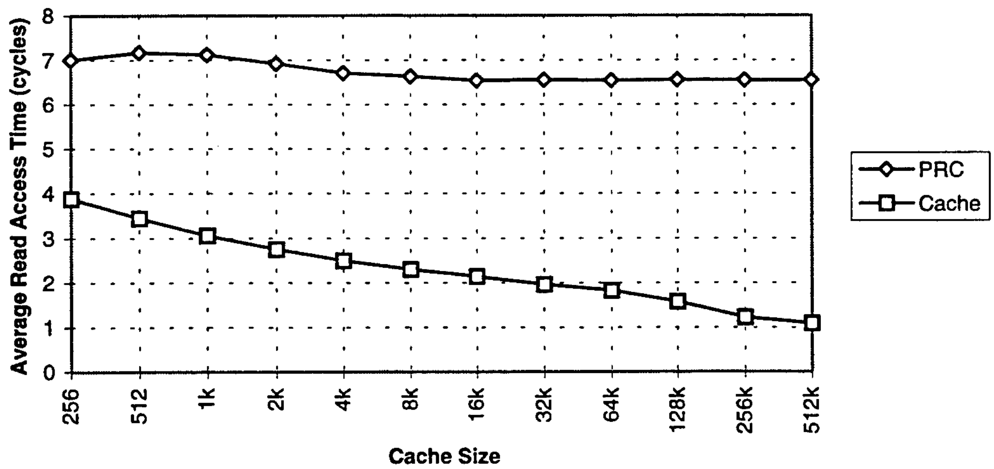


Figure 14. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus80

3. Fully Associative First-level Cache Simulations

The first-level cache simulations were repeated with the same cache sizes but with four-way set associativity. The results are summarized in Figures 15-18.

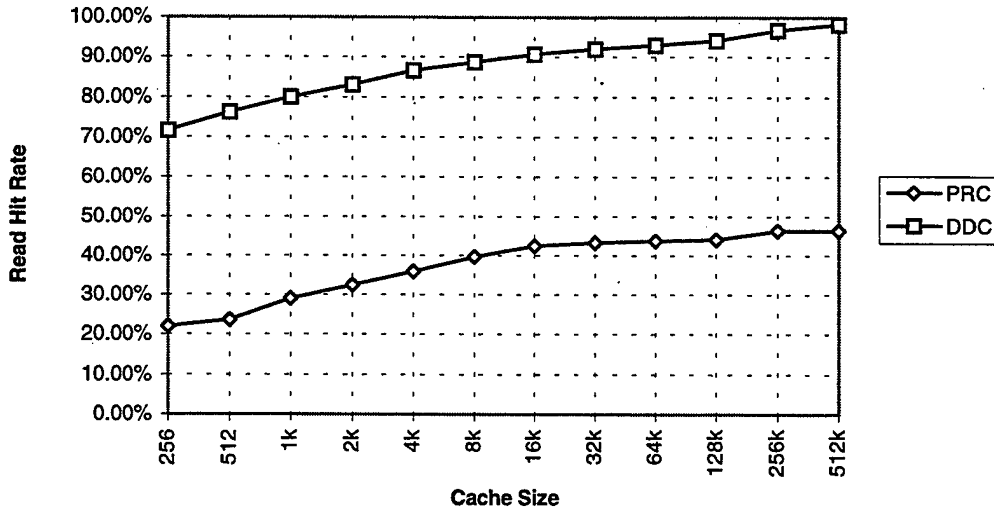


Figure 15. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus20

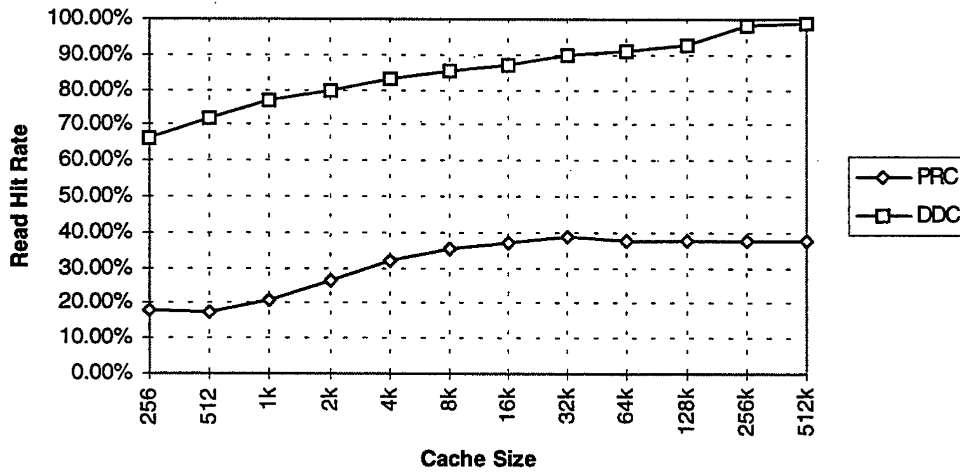


Figure 16. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus80

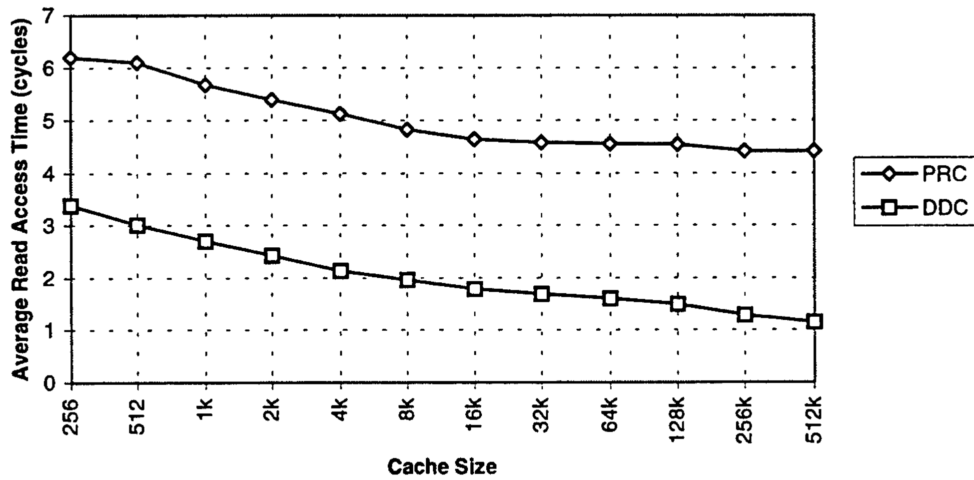


Figure 17. Access Time vs. Cache Size for Fully Associative Cache, Kenbus20

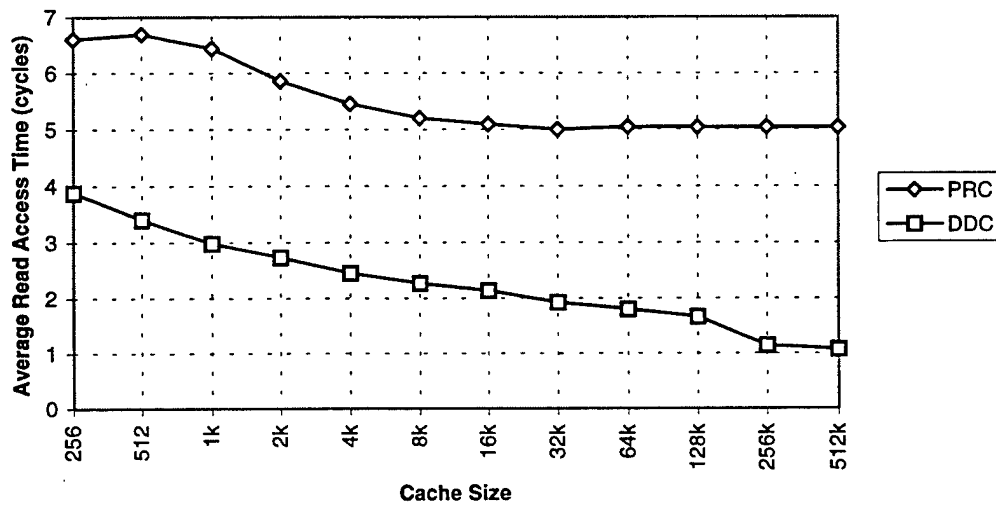


Figure 18. Access Time vs. Cache Size for Fully Associative Cache, Kenbus80

D. TRADITIONAL CACHE VS. PRC SIMULATION CONCLUSIONS

The traditional demand-driven cache performance as a first-level cache far exceeds that of a PRC. The read access times for the demand-driven cache are an average of 2.38 cycles across all associativity types simulated with the Kenbus80 benchmark and 2.14

cycles with the Kenbus20 benchmark. The read access time average for the PRC is 6.36 cycles across all associativity types simulated with the Kenbus80 benchmark and 5.89 cycles with the Kenbus20 benchmark, which is a decrease in performance of over two and a half times. The demand-driven cache average read hit rate across all associativity types simulated with the Kenbus80 benchmark is 84.07% and 86.56% with Kenbus20, while the PRC average read hit rate is 17.86% and 23.19% respectively. Clearly, a first-level cache which is purely predictive in nature is not feasible as a first-level cache.

IV. THE DEVELOPMENT AND SIMULATION OF A DEMAND PRC

The poor performance of the PRC as a first-level cache lead to a comparison of the read miss patterns occurring in the PRC vs. a demand-driven cache. It was determined that a large number of the read misses occurred in the PRC were data addresses that were being accessed frequently but were not part of a data array. When a request for a data address is made of the PRC and that request misses, the predicted data is the only data that is added to the cache. The original request is not put in the cache as it is in a demand-driven cache. During the simulations conducted by Altmisdort [Ref. 5] all original requests were stored in the first-level demand driven cache. Future requests resulted in a read hit in the first-level cache, the PRC (as a second-level cache) was never queried for the data.

The development of a new algorithm was proposed to combine the effects the demand-driven cache and the PRC. The new cache will put both the original request data into the cache as well as the predicted data.

A. FIRST-LEVEL DEMAND PRC CAPSIM CONFIGURATION

Major program changes were required within CaPSim to simulate the new algorithm. The original PRC module only had the capability to store predicted data, not requested data. The changes made to allow the PRC to act as a first-level cache simplified the changes needed to make it a demand PRC.

The distinction of the read requests from the prefetch requests was the first step in storing the demand data. The method for storing the prefetches was already coded into

CaPSim. Those procedures were copied and modified to handle a demand request vice a prefetch request and added to the PRC logic module. The changes were made in such a way as not to interfere with the prediction function of the logic.

B. FIRST-LEVEL DEMAND PRC SIMULATION RESULTS

Figures 19-36 show the simulation results for direct-mapped cache, four-way set-associative cache and fully associative cache, respectively. Read hit rate, average read access times and speed up are indicated.

1. Direct-Mapped First-level Cache Simulations

The direct-mapped first-level cache simulations are conducted with the traditional demand driven cache and the PRC as first-level caches. The first-level cache size is varied between 256 bytes to 512 Kbytes, with each simulation increasing the size by a factor of two. Figures 19-22 summarize the results for the read hit rate and average read access times respectively. Figures 23 and 24 show the speedup of the demand PRC over the traditional demand driven cache as a function of cache size for Kenbus20 and Kenbus80 respectively.

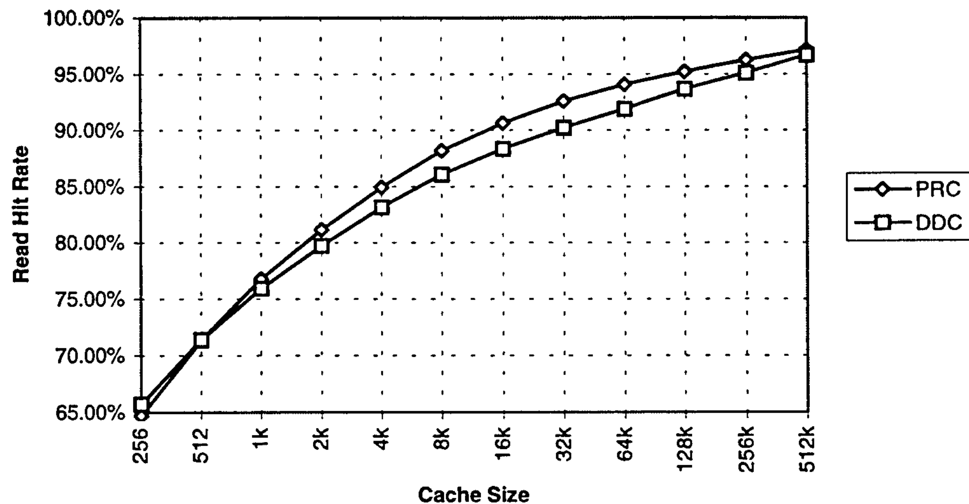


Figure 19. Hit Rate vs. Cache Size for Direct-mapped cache, Kenbus20

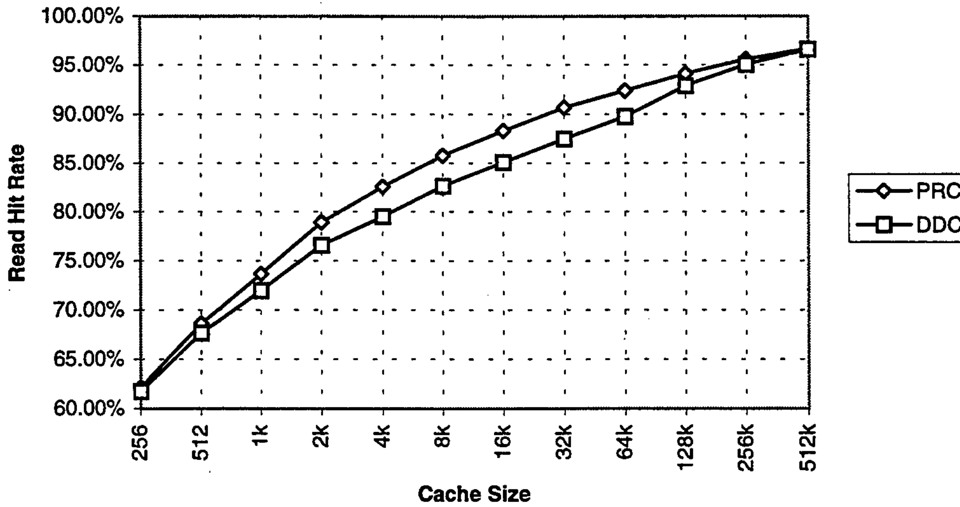


Figure 20. Hit Rate vs. Cache Size for Direct-mapped Cache, Kenbus80

The hit rate for a direct-mapped demand PRC provided an improvement of 0.4% to 3.21% in the Kenbus80 benchmarks and 0.85% to 2.33% with the Kenbus20 benchmarks. An improvement was realized for all cache sizes simulated, with greater improvement demonstrated in the 8Kbyte, 16Kbyte and 32Kbyte cache sizes.

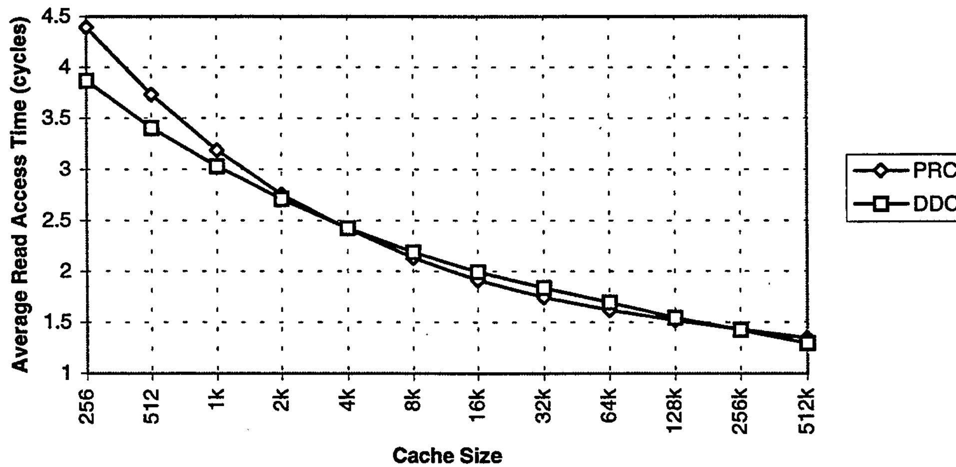


Figure 21. Access Time vs. Cache Size for Direct-mapped Cache, Kenbus20

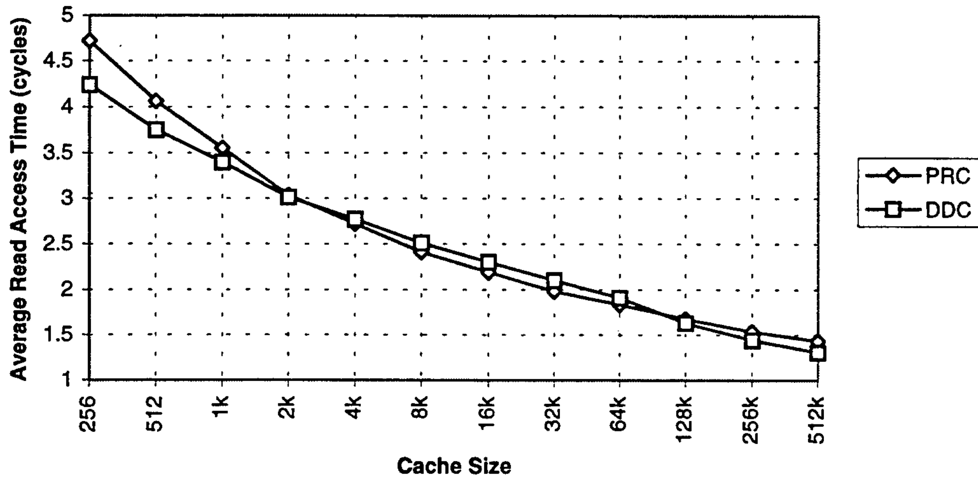


Figure 22. Read Access Time vs. Cache Size for Direct-mapped Cache, Kenbus80

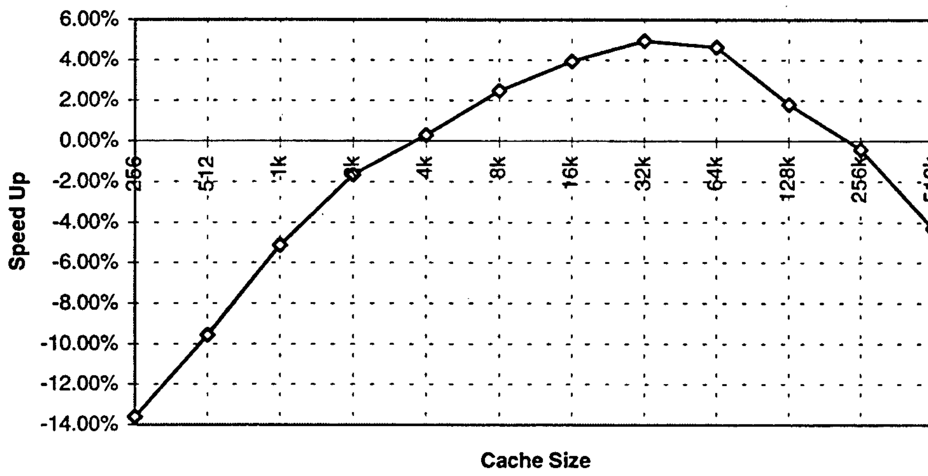


Figure 23. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus20

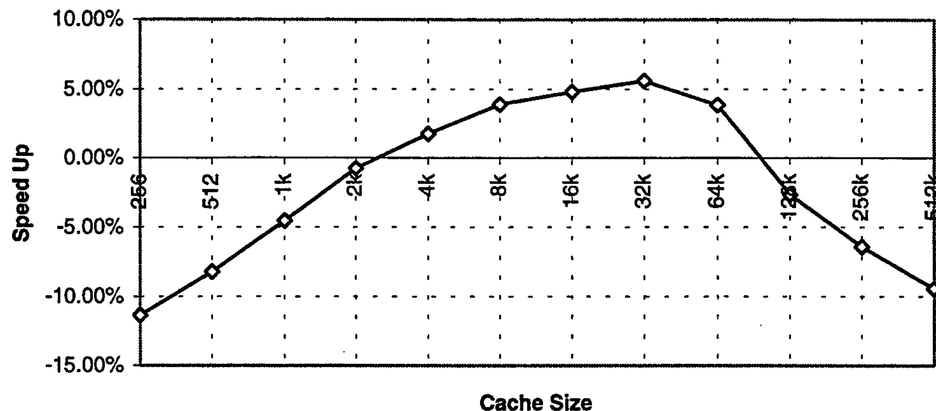


Figure 24. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus80

The speed up of the demand PRC over the traditional demand driven PRC for the direct-mapped case ranges from 1.8% to 5.7%(Kenbus80) and 0.28% to 4.94%(Kenbus20), with the maximum speed up in the 32Kbyte case. For cache sizes of 256 bytes to 4Kbytes and sizes equal to and greater than 128Kbytes, the speedup is negative.

The reason for the bell-shaped speed up curve it two-fold. The speedup is negative in the smaller cache sizes because the cache is attempting to put too many blocks into the cache. Since nearly every CPU request will result in two blocks being placed in the cache (the original request and the prefetch), in the smaller cache sizes the PRC will have more conflict misses than the DDC. Speedup continues to increase until it reaches maximum and then decreases, eventually becoming negative. This occurs because with the larger cache sizes, the bandwidth between the cache and main memory saturates in the PRC case due to the large number of data requests generated.

2. 4-Way Set-associative First-level Cache Simulations

The first-level cache simulations were repeated with the same cache sizes but with 4-way set associativity. The results are summarized in Figures 25-30.

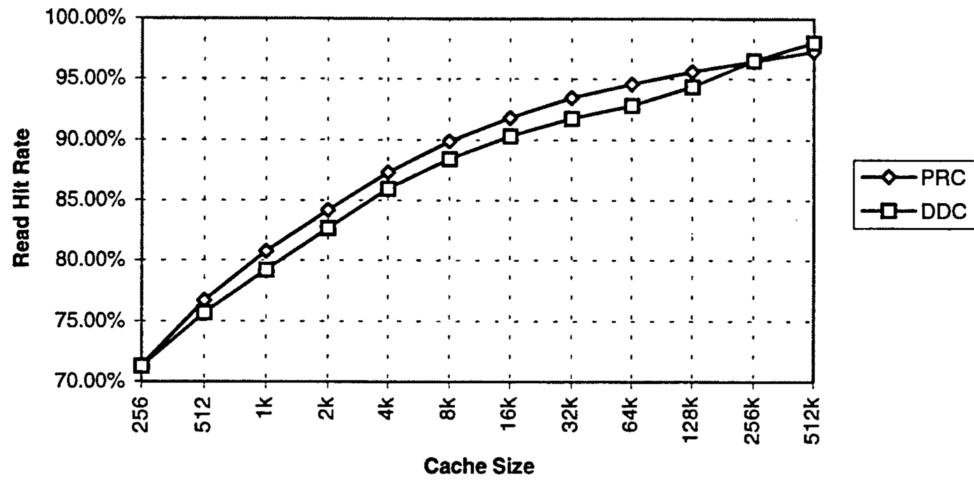


Figure 25. Hit Rate vs. Cache Size for 4-Way Set-associative Cache, Kenbus20

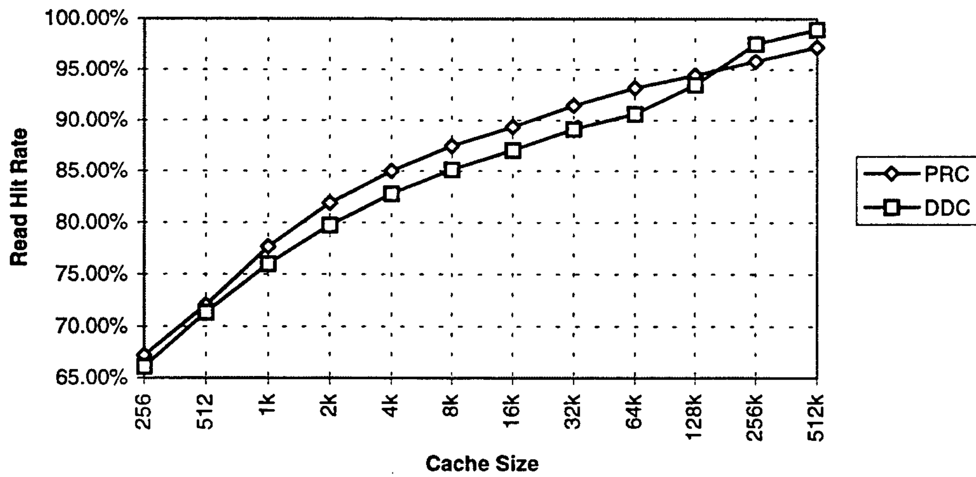


Figure 26. Hit Rate vs. Cache Size for 4-way Set-associative Cache, Kenbus80

The hit rate for a 4-way set-associative demand PRC provided an improvement of 0.7% to 2.6% for cache sizes up to 256Kbytes with the Kenbus80 benchmark and 0.05% to 1.24% for Kenbus20. The greater improvement was again at the 8Kbyte, 16Kbyte and 32Kbyte cache sizes.

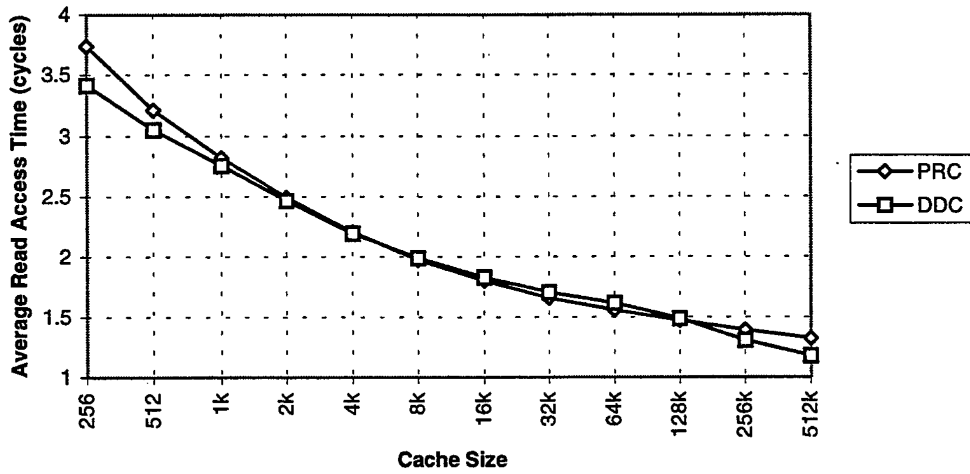


Figure 27. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus20

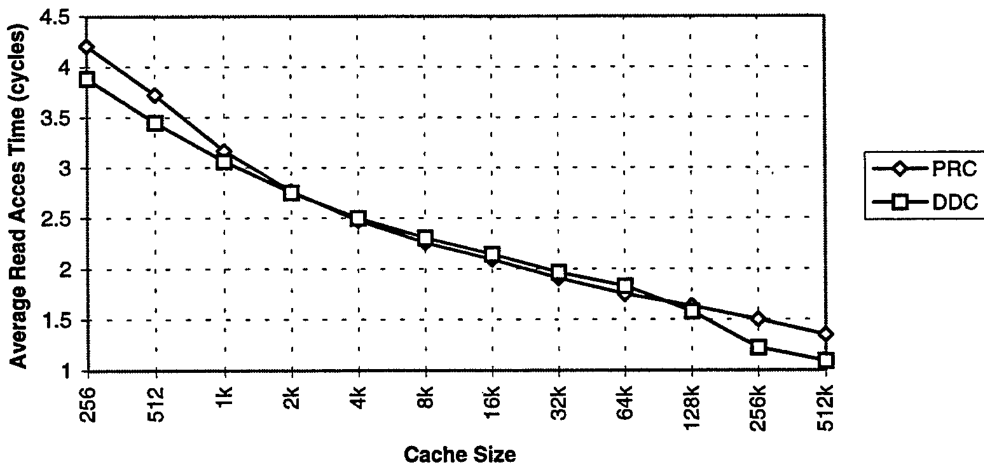


Figure 28. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus80

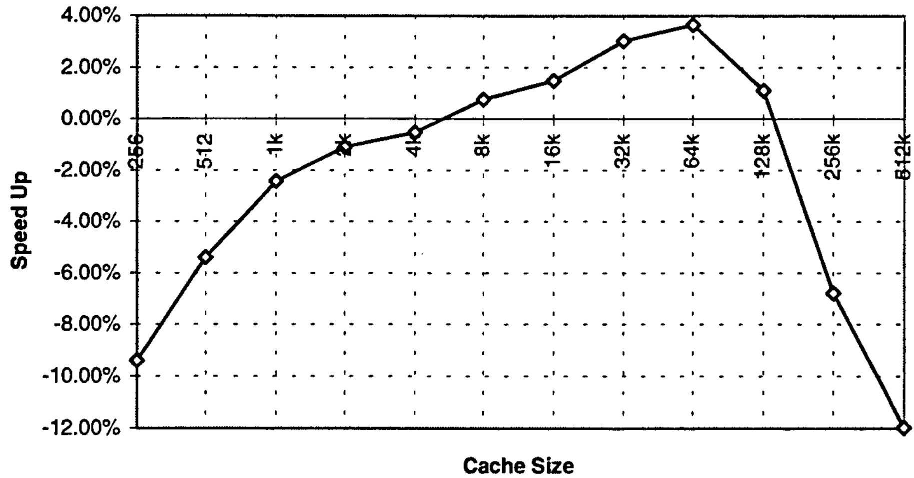


Figure 29. Speed up vs. Cache Size for 4-way Set-associative Cache, Kenbus20

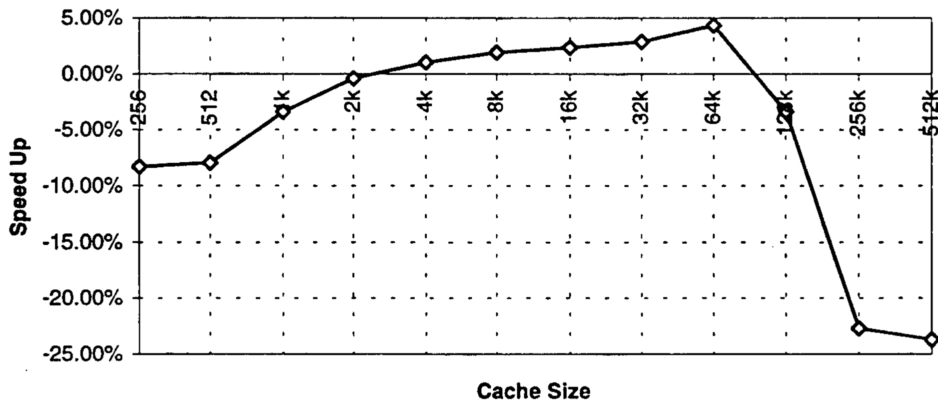


Figure 30. Speed Up vs. Cache Size for 4-Way Set Associative Cache, Kenbus80

The speedup for the 4-way set-associative organization ranges from 1% to 4.3%(Kenbus80) and 0.75% to 3.66%(Kenbus20) with a maximum at a cache size of 64Kbytes. The speedup is negative for cache sizes up to and including 2Kbytes and equal to or greater than 128Kbytes for the Kenbus80 benchmark. With the Kenbus20 benchmark, the speedup is negative for cache sizes up to and including 4Kbytes and cache sizes equal or

greater than 256Kbytes. The 4-way set-associative organization also displays the same bell-shaped speed up curve as the direct-map case.

3. Fully Associative First-level Cache Simulations

The first-level cache simulations were repeated with the same cache sizes but with full associativity. The results are summarized in Figures 31-36.

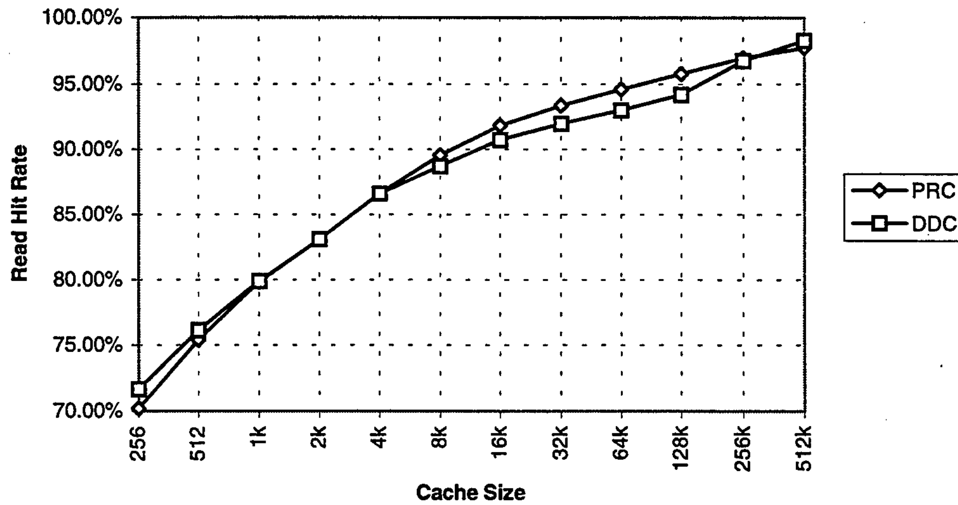


Figure 31. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus20

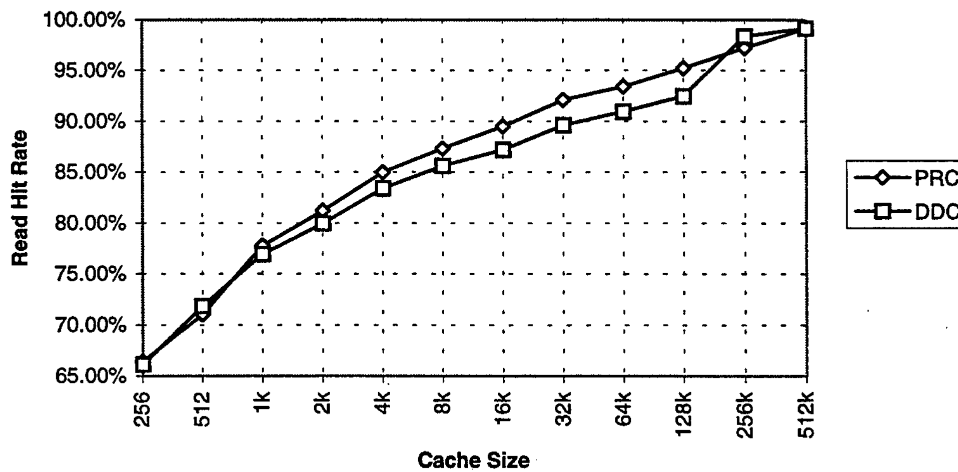


Figure 32. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus80

The hit rate for fully associative case provided an improvement of 0.3% to 2.7% (Kenbus80) and 0.03% to 1.57%(Kenbus20) with greater improvement in cache sizes from 16Kbytes to 128Kbytes.

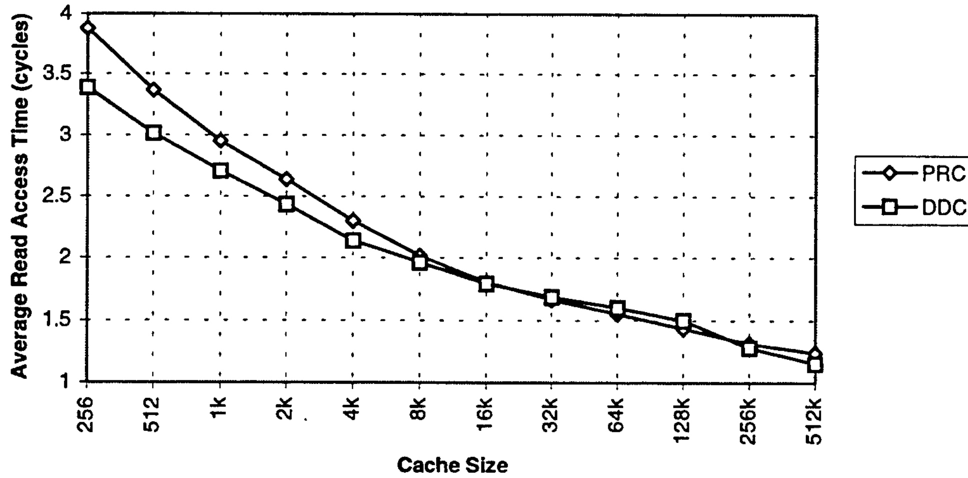


Figure 33. Access Time vs. Cache Size for Fully Associative Cache, Kenbus20

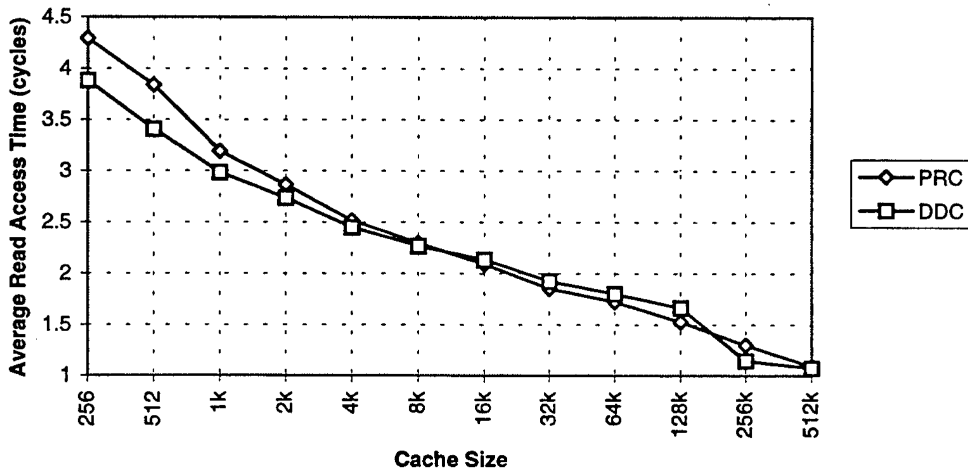


Figure 34. Access Time vs. Cache Size for Fully Associative Cache, Kenbus80

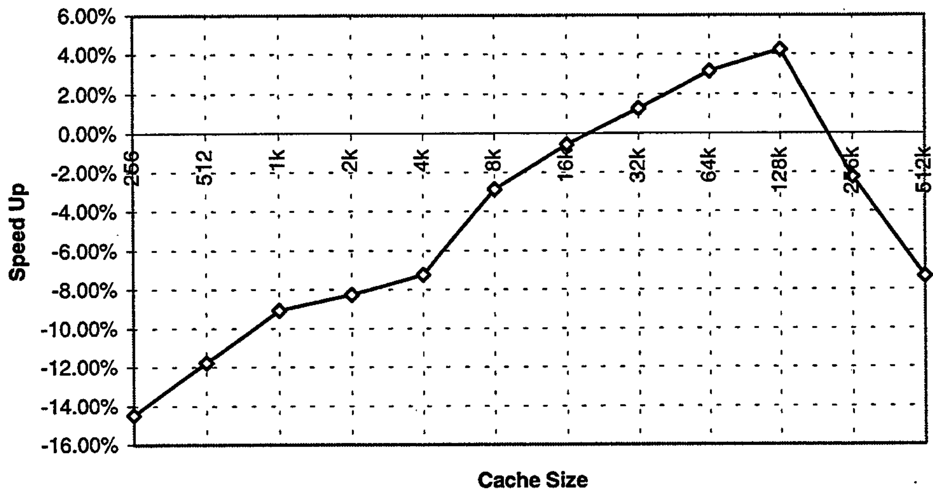


Figure 35. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus20

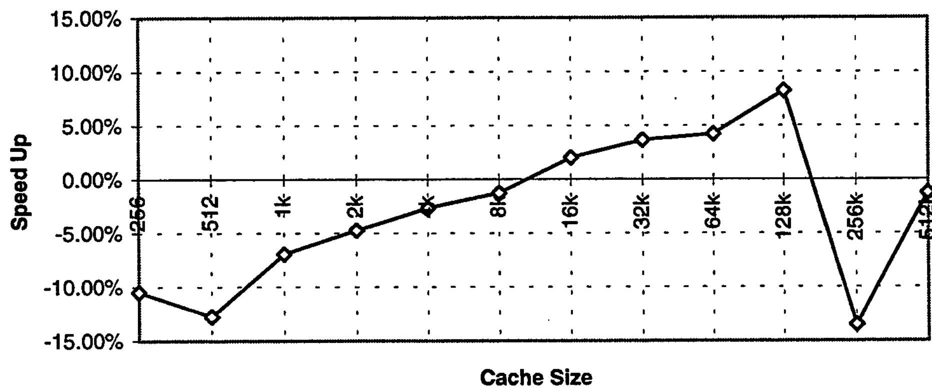


Figure 36. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus80

The speedup for the fully associative organization ranges from 2% to 8.3%(Kenbus80) and 1.24% to 4.23%(Kenbus20), with a maximum speedup at a cache size of 128Kbytes. Negative speedup occurs in cache sizes up to and including 8Kbytes(Kenbus80) 16Kbytes(Kenbus20) and greater than or equal to 256Kbytes. The same general bell-shaped speed up curve is again observed in the fully associative case.

C. FIRST-LEVEL DEMAND PRC CONCLUSIONS

The first-level demand PRC read hit rate is an improvement when compared with the read hit rate of a traditional purely demand-driven cache.

The improvement in the average read access time of the demand PRC was less than that identified in the hit rate. There are instances when the hit rate for the demand PRC is higher than that of the traditional cache but the average read access is higher for the demand PRC. The reason the PRC does not produce any speedup in these cases is due to the stall cycle encountered when the PRC is trying to forward a read request it received from the CPU but the buffer is busy handling a previous request.

The demand PRC demonstrated an improvement in performance in most cases. The most consistent performance improvement was observed in cache sizes ranging from 16Kbytes to 64Kbytes.

V. THE DEVELOPMENT AND SIMULATION OF A PRIORITY-DEMAND PRC

The hit rate improvement of the demand PRC over the purely demand driven cache is quite significant. However, the read access time and overall speedup is not as significant and, in some cases, there is a negative impact. A study of the timing issues revealed that the speedup improvement is hindered by the overload in the Buffer Module caused by the prefetch requests. An improvement of the demand PRC algorithm was developed which prioritizes the buffer tasks and ensures the read requests that originate with the CPU are handled as quickly as possible, even at the price of preempting a prefetch request which is in the process of being transferred.

A. PRIORITY-DEMAND PRC CAPSIM CHANGES

In order for the read requests to be handled in a prioritized order, the Buffer Module of CaPSim was modified. Transactions are assigned a priority based upon the type of transaction: read or prefetch. Transactions of the read type are the CPU requested read data and have the higher priority. Transactions of the prefetch type originate in the PRC module and have the lower priority. The new CaPSim Buffer Module preempts any prefetch transaction when an incoming read request arrives. This ensures the read requests will be completed as expeditiously as possible.

B. FIRST-LEVEL PRIORITY-DEMAND PRC SIMULATION RESULTS

Figures 37-54 show the simulation results for direct-mapped cache, four-way set-associative cache and fully associative cache, respectively. Read hit rate, average read access times and speed up are indicated.

1. Direct-Mapped First-level Cache Simulations

The direct-mapped first-level cache simulations are conducted with the traditional demand driven cache and the PRC as first-level caches. The first-level cache size is varied from 256 bytes to 512 Kbytes, with each simulation increasing the size by a factor of two.

Figures 37-40 summarize the results for the read hit rate and average read access times respectively. Figures 41 and 42 shows the speedup of the demand PRC over the traditional demand driven cache as a function of cache size.

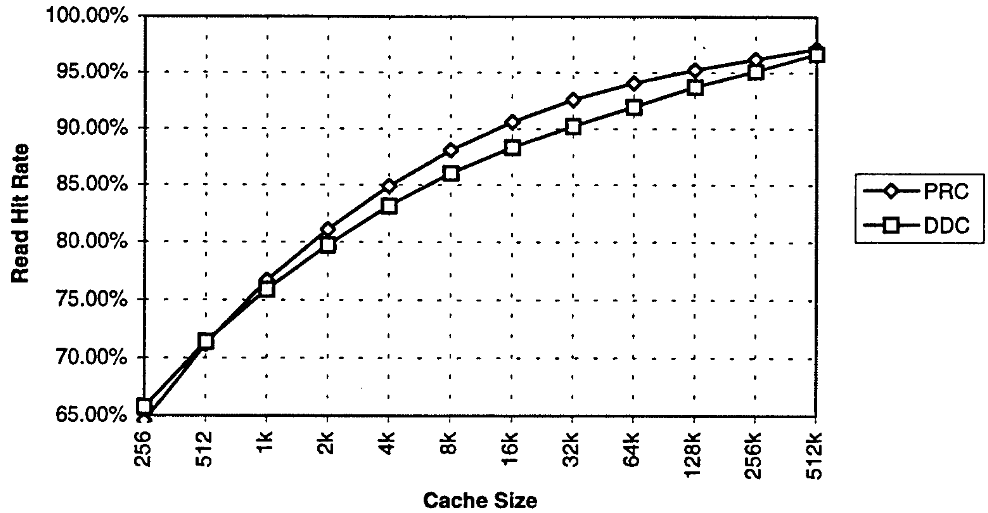


Figure 37. Hit Rate vs Cache Size for Direct-mapped Cache, Kenbus20

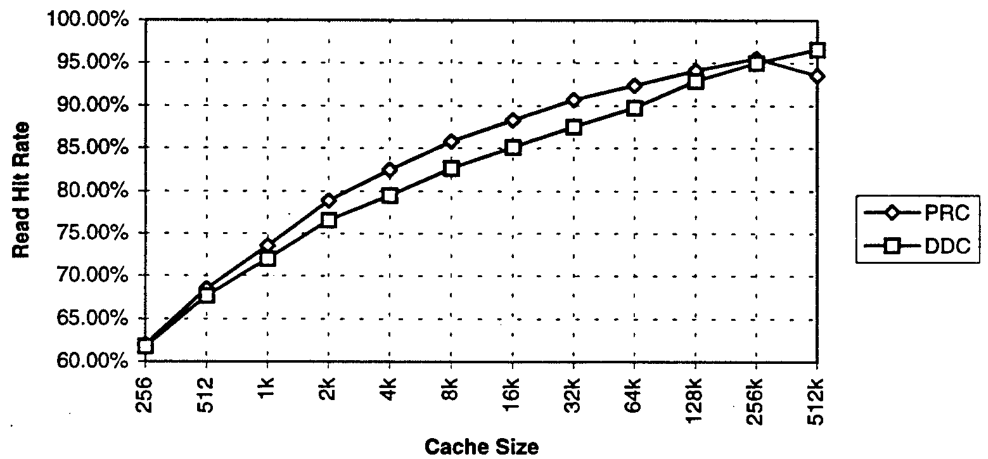


Figure 38. Hit Rate vs. Cache Size for Direct-mapped Cache, Kenbus80

The hit rate for a direct-mapped demand priority PRC provided an improvement of 0.3% to 3.21%(Kenbus80) and 0.77% to 2.3%(Kenbus20) over a demand driven cache. An improvement was recognized through all cache sizes (with the exception of the 512k size

for the Kenbus80 benchmark) simulated with greater improvement demonstrated in the 8Kbyte, 16Kbyte and 32Kbyte cache sizes.

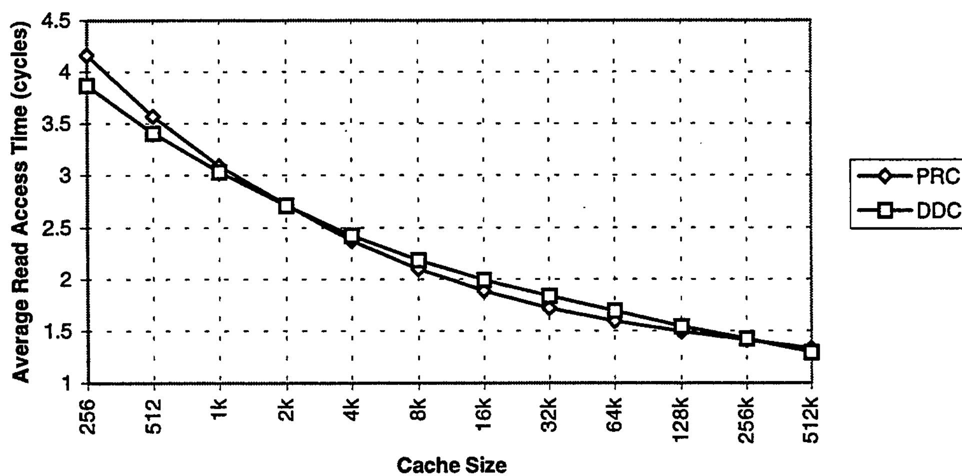


Figure 39. Access Time vs. Cache Size for Direct-mapped Cache, Kenbus20

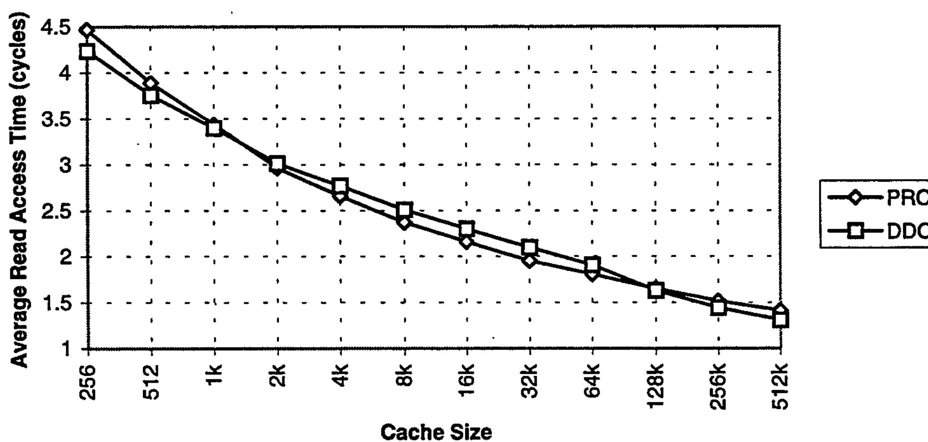


Figure 40. Access Time vs. Cache Size for Direct-mapped Cache, Kenbus80

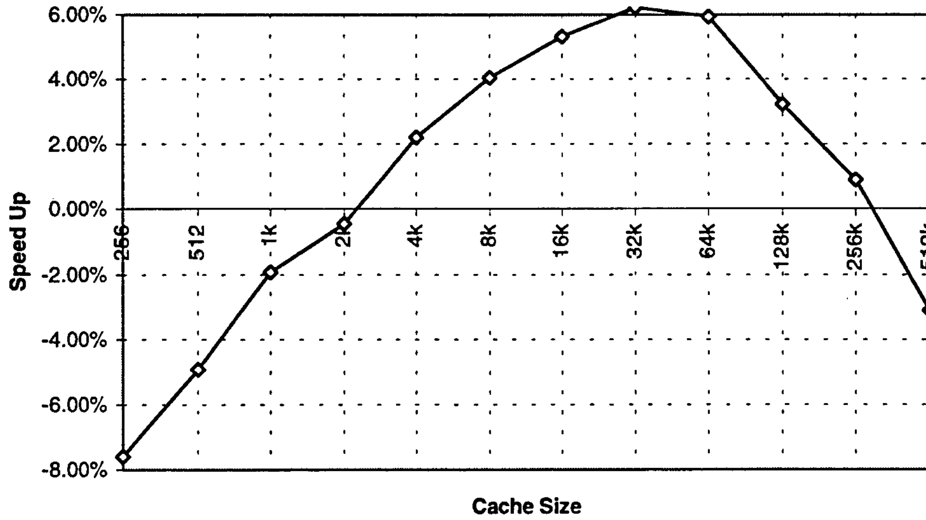


Figure 41. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus20

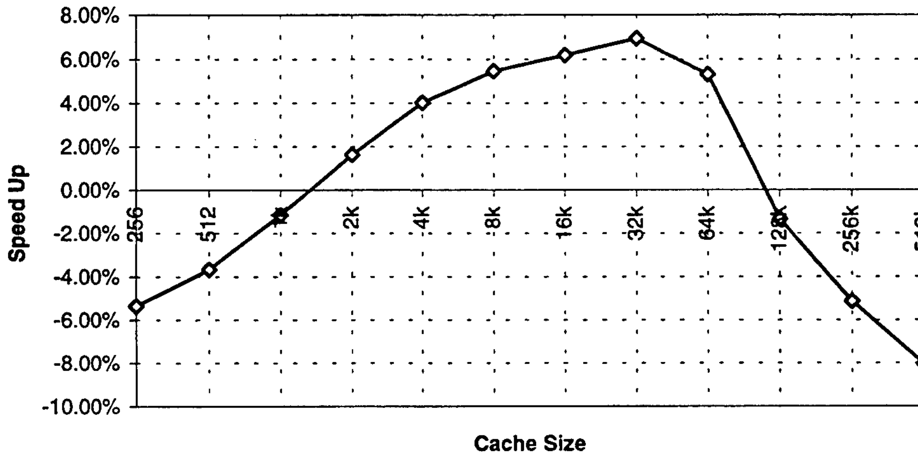


Figure 42. Speed Up vs. Cache Size for Direct-mapped Cache, Kenbus80

The speed up of the priority-demand PRC over the traditional demand driven cache for the direct-mapped case varied from 1.6% to 7%(Kenbus80) and 0.91% to 6.9%(Kenbus20), with the maximum speed up in the 32Kbyte case. For cache sizes of 256 bytes to 1Kbytes(Kenbus80) or 2Kbytes(Kenbus20) and sizes equal to and greater than 128Kbytes(Kenbus80) or 256Kbytes(Kenbus20), the speedup is negative. This speed up

plot maintains the bell-shaped pattern of the direct-mapped demand PRC plot, but the maximum speedup is greater and more cache sizes provide a positive speed up.

2. 4-Way Set-associative First-level Cache Simulations

The first-level cache simulations were repeated with the same cache sizes but with 4-way set associativity. The results are summarized in Figures 43-48.

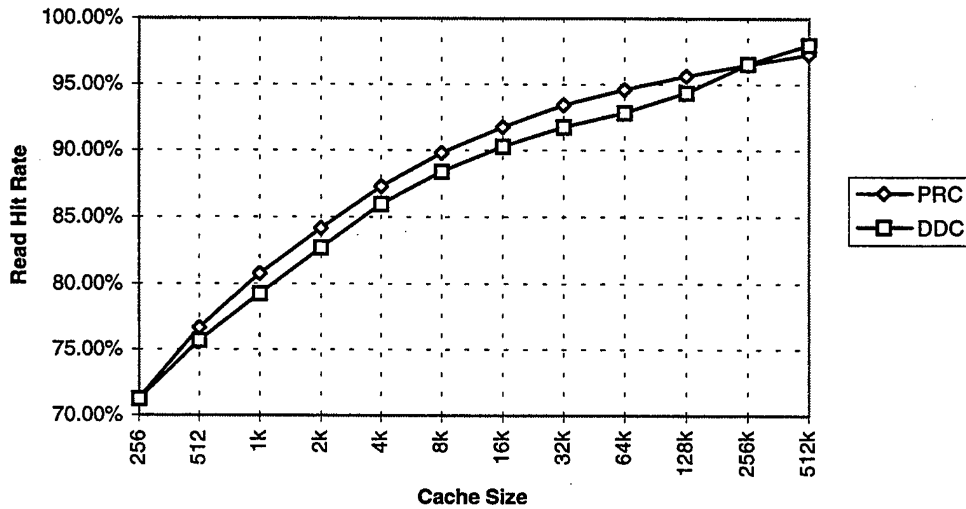


Figure 43. Hit Rate vs. Cache Size for 4-way Set-associative Cache, Kenbus20

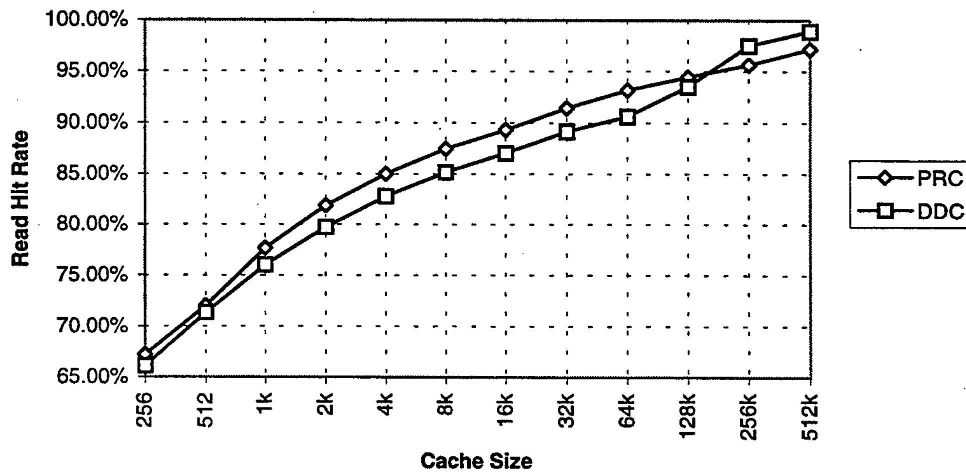


Figure 44. Hit Rate vs. Cache Size for 4-way Set-associative Cache, Kenbus80

The hit rate for a 4-way set-associative priority-demand PRC provided an improvement of 0.7% to 2.6%(Kenbus80) and 0.03% to 1.75%(Kenbus20) for cache sizes up to 256Kbytes. The greater improvement was observed for the 8Kbyte, 16Kbyte, 32Kbyte and 64KByte cache sizes.

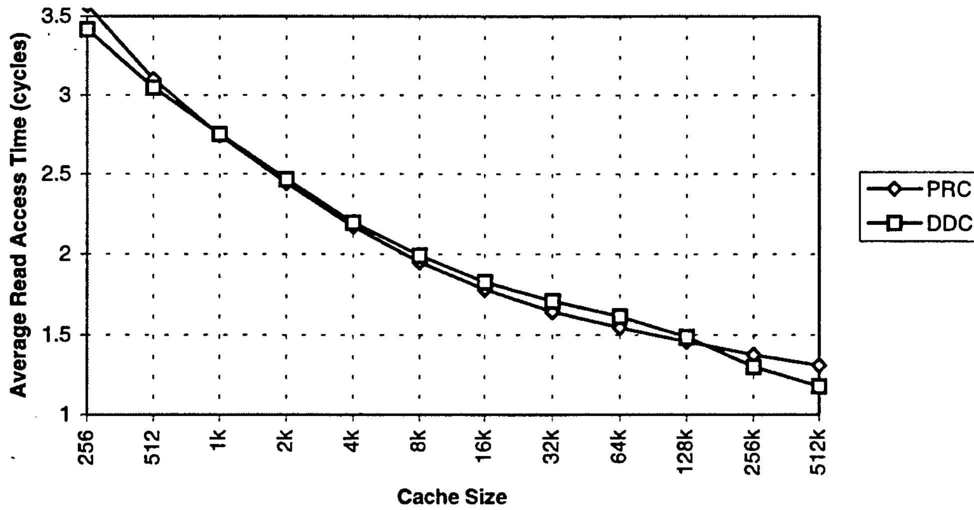


Figure 45. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus20

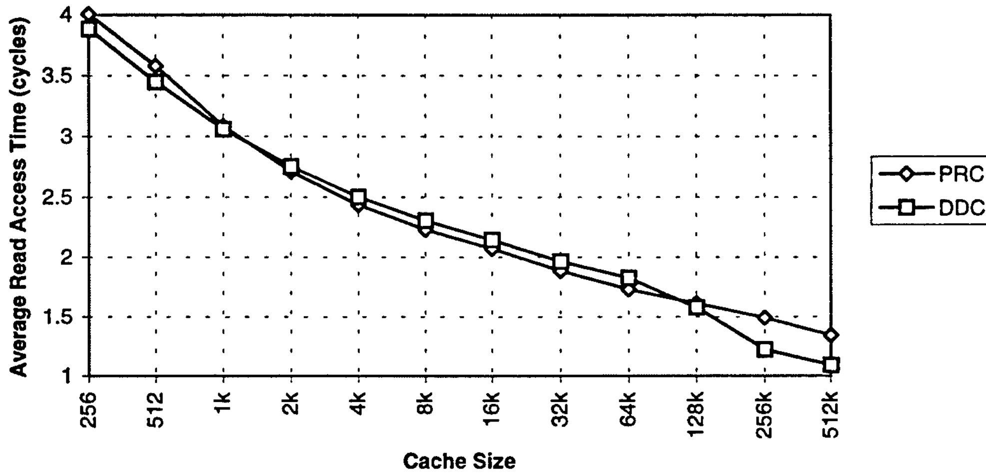


Figure 46. Access Time vs. Cache Size for 4-way Set-associative Cache, Kenbus80

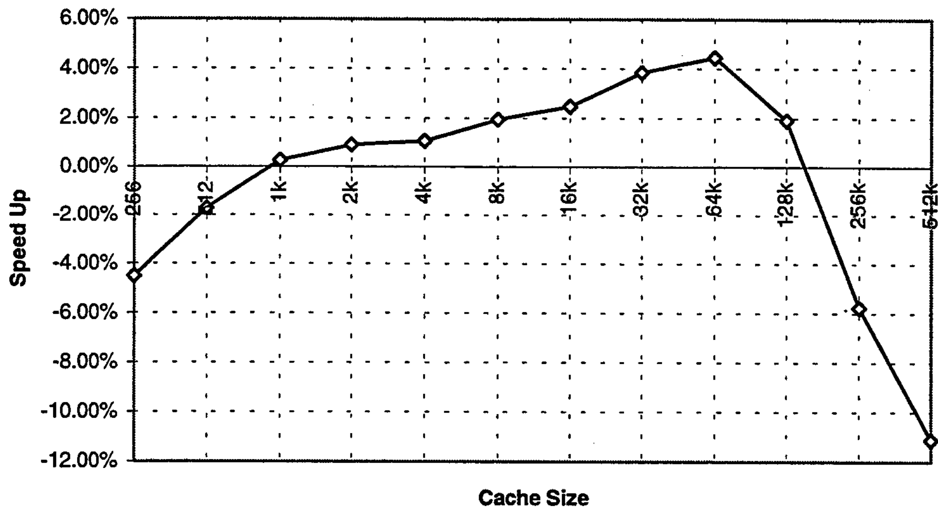


Figure 47. Speed Up vs. Cache Size for 4-Way Set-associative Cache, Kenbus20

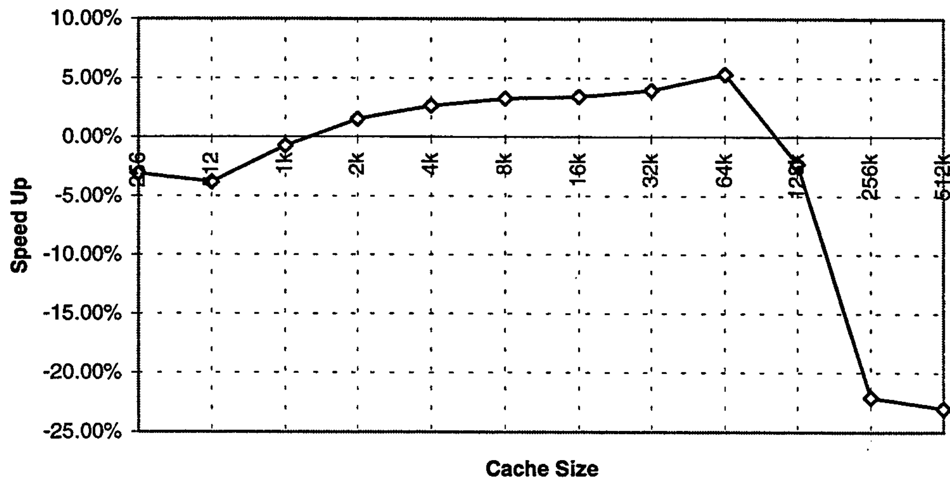


Figure 48. Speed Up vs. Cache Size for 4-Way Set-associative Cache, Kenbus80

The speedup for the 4-way set-associative organization ranges from 1.5% to 5.3%(Kenbus80) and 0.27% to 4.45%(Kenbus20) with a maximum at a cache size of 64Kbytes. The speedup is negative for cache sizes up to and including 512bytes(Kenbus20) or 1Kbytes(Kenbus80) and equal to or greater than 128Kbytes(Kenbus80) or 256Kbytes(Kenbus20). The speedup plot is similar in shape to the 4-way set associative

demand PRC speedup plot in the previous chapter, but the maximum speed up is greater and a wider range of cache sizes generate positive speed up.

3. Fully Associative First-level Cache Simulations

The first-level cache simulations were repeated with the same cache sizes but with full associativity. The results are summarized in Figures 49-54.

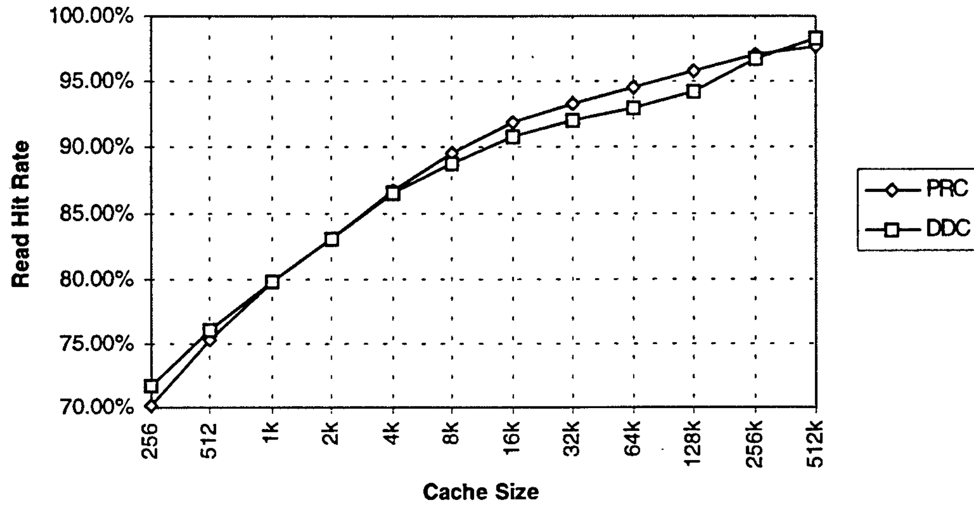


Figure 49. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus20

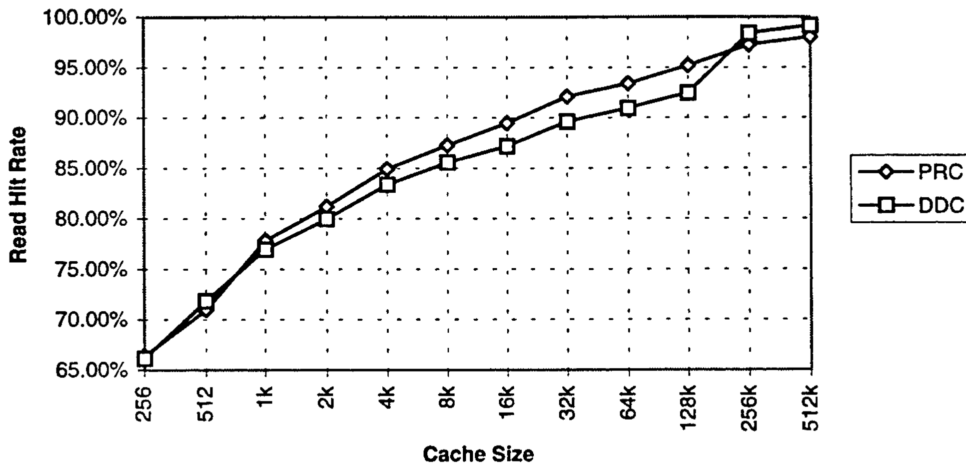


Figure 50. Hit Rate vs. Cache Size for Fully Associative Cache, Kenbus80

The hit rate for the fully associative case provided an improvement of 0.3% to 2.7%(Kenbus80) and 0.05% to 1.56%(Kenbus20), with greater improvement in cache sizes from 16Kbytes to 128Kbytes.

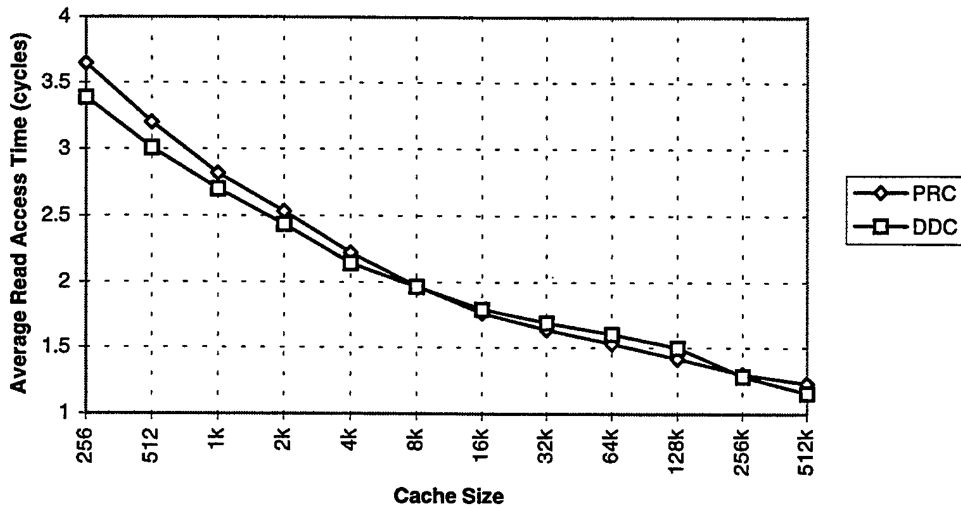


Figure 51. Access Time vs. Cache Size for Fully Associative Cache, Kenbus20

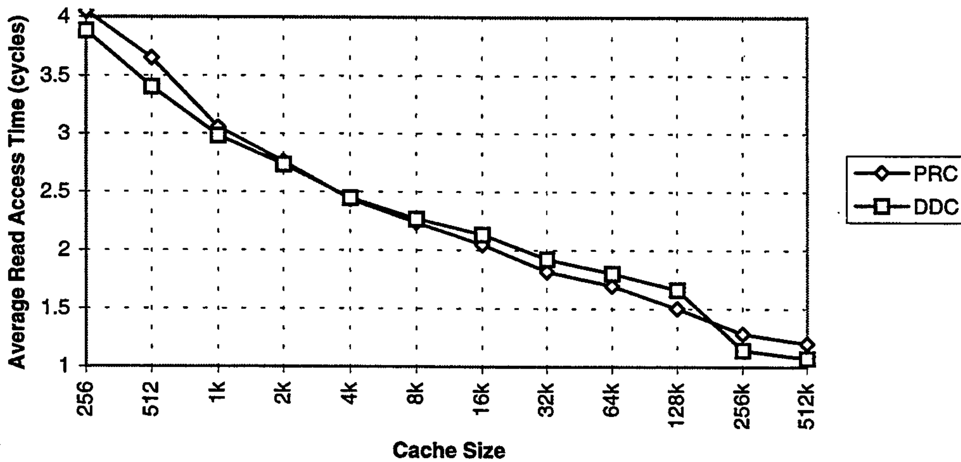


Figure 52. Access Time vs. Cache Size for Fully Associative Cache, Kenbus80

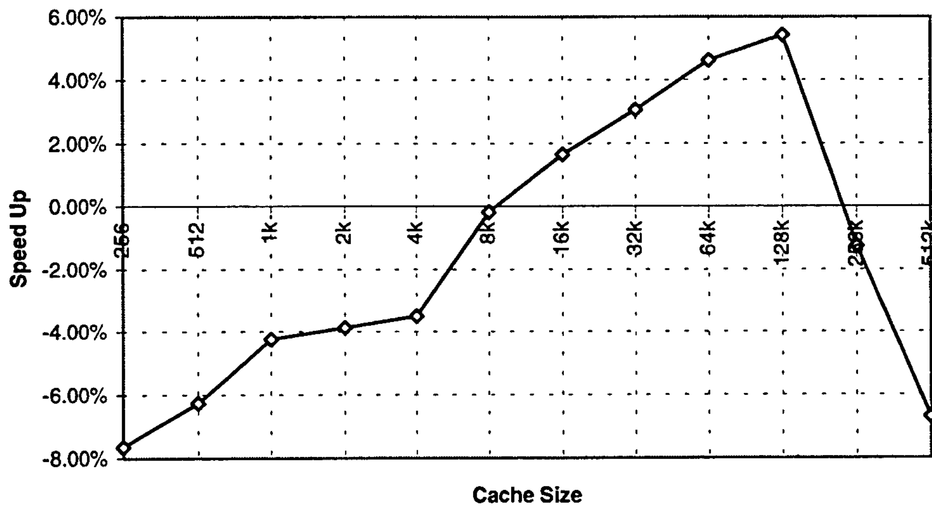


Figure 53. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus20

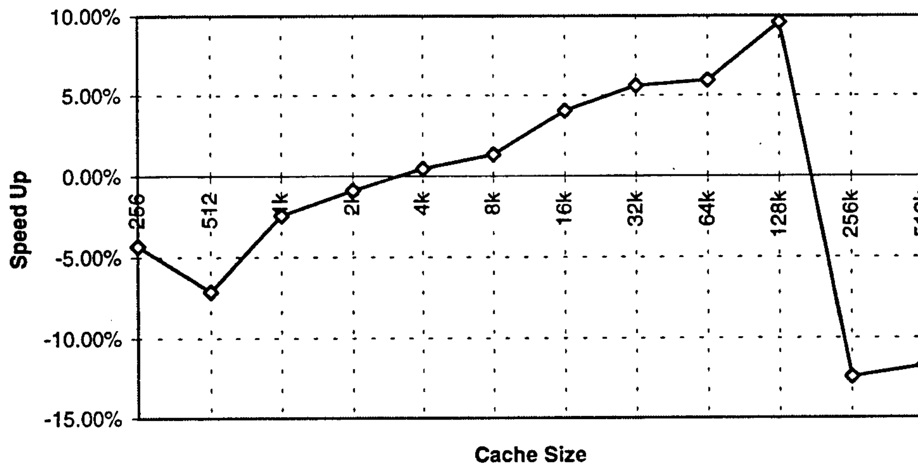


Figure 54. Speed Up vs. Cache Size for Fully Associative Cache, Kenbus80

The speedup for the fully associative organization ranges from .5% to 9.6%(Kenbus80) and 1.24% to 5.42%(Kenbus20), with a maximum speedup at a cache size of 128Kbytes. Negative speedup occurs in cache sizes up to and including 2Kbytes(Kenbus80) or 8Kbytes(Kenbus20) and greater than or equal to 256Kbytes. The 20% drop in speed up observed in the Kenbus80 benchmark from 128Kbytes to 256Kbytes seems to be a factor in the DDC's response to the benchmark. The PRC's read access times

remain smooth but the DDC has a large decrease in read access time and, correspondingly, a large jump in the hit rate between 128Kbyte and 256Kbyte cache sizes.

C. FIRST-LEVEL PRIORITY-DEMAND PRC CONCLUSIONS

The first-level priority-demand PRC read-hit rate is an improvement when compared with the read-hit rate of a traditional purely demand-driven cache.

The improvement in the average read access time of the priority-demand PRC was much better than that demonstrated in the demand PRC. The priority preemption of tasks in the buffer module successfully lowered the average read-access rate.

The priority-demand PRC demonstrated an improvement in performance in the majority of cache sizes. The most consistent performance improvement was observed in cache sizes ranging from 16Kbytes to 64Kbytes.

VI. CONCLUSIONS

A. EFFECTIVENESS OF THE PRC AS A FIRST-LEVEL CACHE

In this thesis, the Predictive Read Cache was accurately simulated as a first-level cache. CaPSim simulation results for both the PRC algorithm and a traditional demand-driven cache were presented. The poor performance of the PRC as a first-level cache lead to the development of a demand PRC which was shown by simulation to have a much higher performance than the original PRC.

The hit rate performance of the demand PRC was higher than that of a traditional cache, but it was felt that the overall speedup could be improved. By designing the buffer module to preempt prefetch transactions in progress, the speedup was improved. The priority-demand PRC dramatically increased the performance of the first-level cache.

B. SUGGESTION FOR FUTURE DEVELOPMENT

The performance of the PRC as a first-level cache can be investigated further by simulating larger address traces of different types. In particular, the new SPEC 98 benchmarks will be available soon and will provide longer address traces to more accurately simulate the performance of the PRC. Different types of address traces, such as those from the SPEC suite rather than the SDM suite, will more accurately reflect the scientific, vice multitasking, environment, for which the PRC is intended.

A larger set of design alternatives can also be simulated. Experimenting with block sizes and different types of set associativity may reveal an optimal configuration for the

memory hierarchy with a PRC. The CaPSim cost analysis tool can be further developed and used to evaluate the cost-performance trade-off of the PRC as a first-level cache.

APPENDIX A. AN EXAMPLE CAPSIM CONFIGURATION FILE

The following is an example of a configuration file used for the simulations of the Predictive Read Cache as a first-level cache:

```
# -------------------------------------------
# CaPSim Configuration File
# Author  : K. Christensen
# Revised : 28 OCT 97
# -------------------------------------------

simulation
{
        Word Size               = 4
        Input  Path             = /data_tehe/altmisdo/Kenbus80/output/
        Output Path             = iPRC_64k/
        Trace Type              = PRC
        Trace Filename          = skenPRC.*****
        Start File Number       = 0
        Stop  File Number       = 99
        Trace Buffer Size       = 10000
        User E-mail Address     = kschrist@nps.navy.mil
}

hierarchy
{
        prc             PRC
        buffer          Buffer1
        memory          MainMemory
}


module PRC
{
        Prediction Algorithm            = Instruction Address Displacement
        PRC size                        = 65536
        Block Size                      = 16
        Associativity                   = *
        SubBlock Size                   = 4
        Replacement Policy              = LRU
        Write Policy                    = Write Through
        Access Time                     = 1
```

```
        Block Buffer Transfer Time    = 1
        Bypass Write Allocates        = Yes
        Maximum read slips in buffer = 2
        Minimum read size  in buffer  = 12
}

module Buffer1
{
        Read  Buffer Size             = 8
        Write Buffer Size             = 4
        Write Buffer Block Size       = 16
        Enforce Priorities            = Yes
        Remove Duplicates             = Yes
}

module MainMemory
{
        Access   Time  = 5
        Transfer Time  = 1
        Transfer Size  = 4
}
```

APPENDIX B. AN EXAMPLE CAPSIM CONFIGURATION FILE

The following is an example of a configuration file used for the simulations of a traditional demand driven cache as a first-level cache.

```
# -------------------------------------------
# CaPSim Configuration File
# Author  : Kathryn Christensen
# Revised : March 10 , 1998
# -------------------------------------------

simulation
{
        Word Size               = 4
        Input  Path             = /data_tehe/camligun/Kenbus80/input/
        Output Path             = L1_64k/
        Trace Type              = BYU
        Trace Filename          = sken.*****
        Start File Number       = 0
        Stop  File Number       = 99
        Trace Buffer Size       = 1000
        User E-mail Address     = kschrist@nps.navy.mil
}

hierarchy
{
        cache           CacheL1
        buffer          Buffer1
        memory          MainMemory
}

module CacheL1
{
        Cache Size              = 65536
        Block Size              = 16
        SubBlock Size           = 4
        Fetch Size              = 16
        Transfer Size           = 4
        Associativity           = *
        Replacement Policy      = LRU
        Write Policy            = Write Through
```

```
        Write Miss Policy            = Write Around
        Wrapping Fetch Policy        = Wrap Up
        Access Time                  = 1
        Read Hit Time                = 0
        Read Miss Time               = 0
        Write Hit Time               = 0
        Write Miss Time              = 0
        Read Forward                 = No
        Enable Block Buffer          = Yes
        Search Block Buffer          = Yes
        Block Buffer Transfer Time   = 1
}


module Buffer1
{
        Read  Buffer Size            = 8
        Write Buffer Size            = 4
        Write Buffer Block Size      = 16
        Enforce Priorities           = Yes
        Remove Duplicates            = Yes
}

module MainMemory
{
        Access   Time  = 5
        Transfer Time  = 1
        Transfer Size  = 4
}
```

APPENDIX C. AN EXAMPLE CAPSIM LOG FILE

```
+--------------------------------------------------------------+
| CaPSim Log File                        F. Nadir ALTMISDORT |
| Sat May  2 01:52:09 1998                                     |
+--------------------------------------------------------------+

Starting configuration ----------------------------------------

CPU        Reading Configuration File ...              : [OK]
CPU        Checking Syntax ...                         : [OK]
CPU        Setting Simulation Parameters ...           : [OK]
CPU        Checking Memory Hierarchy ...               : [OK]
CPU        Checking Input/Output Paths ...             : [OK]
CPU        Starting Self-Test ...                      : [OK]

Initializing simulation module CacheL1                 : [ 1]
CacheL1     Cache Size                                 : [OK]
CacheL1     Block Size                                 : [OK]
CacheL1     SubBlock Size                              : [OK]
CacheL1     Fetch Size                                          : [OK]
CacheL1     Transfer Size                              : [OK]
CacheL1     Associativity                              : [OK]
CacheL1     Replacement Policy                         : [OK]
CacheL1     Write Policy                               : [OK]
CacheL1     Write Miss Policy                          : [OK]
CacheL1     Wrapping Fetch Policy                      : [OK]
CacheL1     Access Time                                : [OK]
CacheL1     Read Hit Time                              : [OK]
CacheL1     Read Miss Time                             : [OK]
CacheL1     Write Hit Time                             : [OK]
CacheL1     Write Miss Time                            : [OK]
CacheL1     Read Forward                               : [OK]
CacheL1     Enable Block Buffer                        : [OK]
CacheL1     Search Block Buffer                        : [OK]
CacheL1     Block Buffer Transfer Time                 : [OK]
CacheL1     Starting Self-Test ...                     : [OK]

Initializing simulation module Buffer1                 : [ 2]
Buffer1    Read  Buffer Size                           : [OK]
Buffer1    Write Buffer Size                           : [OK]
Buffer1    Write Buffer Block Size                     : [OK]
```

```
Buffer1     Enforce Priorities                              : [OK]
Buffer1     Remove Duplicates                               : [OK]
Buffer1     Starting Self-Test ...                          : [OK]

Initializing simulation module MainMemory                   : [ 3]
MainMemory    Access   Time                                 : [OK]
MainMemory    Transfer Time                                 : [OK]
MainMemory    Transfer Size                                 : [OK]
MainMemory    Starting Self-Test ...                        : [OK]

Finalizing simulation modules ...                           :
CPU           Finalize ...                                  : [OK]
CacheL1       Finalize ...                                  : [OK]
Buffer1       Finalize ...                                  : [OK]
MainMemory    Finalize ...                                  : [OK]

CaPSim configuration completed successfully @ Sat May  2 01:52:10 1998

Starting simulation -----------------------------------------------
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00000 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00001 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00002 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00003 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00004 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00005 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00006 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00007 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00008 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00009 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00010 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00011 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00012 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00013 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00014 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00015 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00016 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00017 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00018 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00019 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00020 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00021 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00022 : [OK]
Opening file  /data_tehe/camligun/Kenbus80/input/sken.00023 : [OK]
```

Opening file /data_tehe/camligun/Kenbus80/input/sken.00024 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00025 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00026 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00027 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00028 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00029 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00030 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00031 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00032 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00033 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00034 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00035 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00036 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00037 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00038 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00039 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00040 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00041 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00042 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00043 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00044 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00045 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00046 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00047 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00048 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00049 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00050 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00051 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00052 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00053 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00054 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00055 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00056 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00057 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00058 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00059 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00060 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00061 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00062 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00063 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00064 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00065 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00066 : [OK]

Opening file /data_tehe/camligun/Kenbus80/input/sken.00067 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00068 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00069 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00070 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00071 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00072 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00073 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00074 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00075 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00076 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00077 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00078 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00079 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00080 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00081 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00082 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00083 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00084 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00085 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00086 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00087 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00088 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00089 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00090 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00091 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00092 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00093 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00094 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00095 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00096 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00097 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00098 : [OK]
Opening file /data_tehe/camligun/Kenbus80/input/sken.00099 : [OK]

The simulation is completed successfully @ Sat May 2 04:23:58 1998

Dumping simulation modules ... :
CPU Dumping L1_64k/CPU_dump.00099 : [OK]
CacheL1 Dumping L1_64k/CacheL1_dump.00099 : [OK]
Buffer1 Dumping L1_64k/Buffer1_dump.00099 : [OK]
MainMemory Dumping L1_64k/MainMemory_dump.00099 : [OK]

Closing Log File .... @ Sat May 2 04:23:59 1998

APPENDIX D. AN EXAMPLE OUTPUT FILE FOR THE CPU MODULE

```
+------------------------------------------------------------------+
| Module Title   : CPU                                             |
| Module ID      : 0                                               |
| Configuration  : L1_64k           Sat May  2 04:23:58 1998 |
+------------------------------------------------------------------+

System Clock : 0071595011 ---------------------------------------------

Operating Parameters ---------------------------------------------

Number of Simulation Modules      : 4
Word Size                         : 4
Trace Type                        : BYU Trace
Trace Filename                    : /data_tehe/camligun/Kenbus80/input/sken.00099
Start File Number                 : 0
Stop  File Number                 : 99
Maximum Trace Buffer Size         : 1000
Current Trace Buffer Index        : 928
Last Entry in Trace Buffer        : 928

Simulation Set ---------------------------------------------------

+------------------+---+
| CPU              | 0 |
+------------------+---+
| CacheL1          | 1 |
+------------------+---+
| Buffer1          | 2 |
+------------------+---+
| MainMemory       | 3 |
+------------------+---+

Event Queue Contents ---------------------------------------------

+----------------------------+
| CaPSim    Event Queue      |
| Size: 00 @ 0071595011      |
+----------------------------+

Number of Canceled Events : 0
```

Module States --

CPU	State @0071595011 : ReadStall
CacheL1	State @0071595011 : Idle Block Buffer : Idle
Buffer1	State @0071595011 : Idle
MainMemory	State @0071595011 : Idle

Statistics --

Total Number of Requests	: 7122928
Total Number of Read Requests	: 4901106
Total Number of Write Requests	: 2221822
Total Read Stall Cycles	: 8829458
Total Write Stall Cycles	: 2221825
Average Read Access Time	: 1.80152357
Average Write Access Time	: 1.00000131

END OF FILE [L1_64k/CPU_dump.00099] ---------------------------------------

APPENDIX E. AN EXAMPLE OUTPUT FILE FOR THE CACHE MODULE

```
+----------------------------------------------------------------+
| Module Title   : CacheL1                                        |
| Module ID      : 1                                              |
| Configuration  : L1_64k            Sat May  2 04:23:58 1998     |
+----------------------------------------------------------------+

System Clock : 0071595011 -------------------------------------------

Operating Parameters ------------------------------------------------

Cache Size                  : 65536
Block Size                  : 16
Sub-Block Size              : 4
Fetch Size                  : 16
Transfer Size               : 4
Associativity               : 4096 (Fully associative)
Number of Sets              : 1
Total Number of Blocks      : 4096
Number of Sub-Blocks        : 4
Replacement Policy          : LRU
Write Policy                : Write Through
Write Miss Policy           : Write Around
Wrapping Fetch Policy       : Wrap Up
Start Policy                : Cold Start
Read Forward                : No
Enable Block Buffer         : Yes
Search Block Buffer         : Yes
Read  Access Time           : 1
Write Access Time           : 1
Read  Hit   Time            : 0
Read  Miss  Time            : 0
Write Hit   Time            : 0
Write Miss  Time            : 0
Block Buffer Transfer Time  : 1

Address Decoder ----------------------------------------------------

+----------------------------+--+--+ +---------------+
|3322222222221111111111000000|00|00| |t : tag  bits = 28|
|1098765432109876543210987654|32|10| |s : set  bits = 00|
```

```
+----------------------------+--+--+ |w : word bits = 02|
|tttttttttttttttttttttttttttt|ww|bb| |b : byte bits = 02|
+----------------------------+--+--+ +------------------+

Block Address Mask          : fffffff0 hex
Sub-block Address Mask      : fffffffc hex
Word Address Mask           : fffffffc hex
Set Number  Mask            : 00000000 hex
Sub-block Number  Mask      : 0000000c hex
Word Number Mask            : 0000000c hex
Word Byte Number  Mask      : 00000003 hex
Block Byte Number Mask      : 0000000f hex

Statistics ------------------------------------------------------------

Total Number Of Read  Requests      : 4901106
Total Number Of Write Requests      : 2221822
Number Of Read  Requests            : 4901106
Number Of Write Requests            : 2221822
Number Of Read  Cancels             : 0
Number Of Write Cancels             : 0
Number Of Read  Hits                : 4457572
Number Of Write Hits                : 1470170
Number Of Dirty Read  Misses        : 0
Number Of Dirty Write Misses        : 0

Global Read  Hit  Ratio             : 0.90950328
Global Read  Miss Ratio             : 0.09049672

Global Write Hit  Ratio             : 0.66169566
Global Write Miss Ratio             : 0.33830434

Local  Read  Hit  Ratio             : 0.90950328
Local  Read  Miss Ratio             : 0.09049672

Local  Write Hit  Ratio             : 0.66169566
Local  Write Miss Ratio             : 0.33830434

Dirty Read  Miss Ratio              : 0.00000000
Dirty Write Miss Ratio              : 0.00000000
Dirty Read  Miss Percentage         : 0.00000000 %
Dirty Write Miss Percentage         : 0.00000000 %
```

Read Miss Cycles	: 4256988
Read Miss Penalty	: 9.59788418
Block Buffer Read Hits	: 0
Block Buffer Write Hits	: 0
Block Buffer Read Hit Ratio	: 0.00000000
Block Buffer Write Hit Ratio	: 0.00000000

END OF FILE [L1_64k/CacheL1_dump.00099] ------------------------------------

APPENDIX F. AN EXAMPLE OUTPUT FILE FOR THE PRC MODULE

```
+----------------------------------------------------------------+
| Module Title   : PRC                                           |
| Module ID      : 1                                             |
| Configuration  : iPRC_64k          Fri Apr 24 14:31:48 1998    |
+----------------------------------------------------------------+

System Clock : 0091740833 --------------------------------------------

Operating Parameters ---------------------------------------------------

PRC Algorithm                  : Instruction Address Displacement
PRC Size                       : 65536
Block Size                     : 16
Sub-Block Size                 : 4
Fetch Size                     : 16
Transfer Size                  : 4
Associativity                  : 4096 (Fully associative)
Number of Sets                 : 1
Total Number of Blocks         : 4096
Number of Sub-Blocks           : 4
Replacement Policy             : LRU
Write Policy                   : Write Through
Write Miss Policy              : Write Around
Bypass Write Allocates         : Yes
Read  Access Time              : 1
Write Access Time              : 1
Read  Hit    Time              : 0
Read  Miss   Time              : 0
Write Hit    Time              : 0
Write Miss   Time              : 0
Block Buffer Transfer Time     : 1

Address Decoder ---------------------------------------------------------

INSTRUCTION ADDRESS DECODER :

+------------------------------+--+ +-----------------+
|332222222222111111111100000000|00| |t : tag  bits = 30|
|109876543210987654321098765432|10| |s : set  bits = 00|
+------------------------------+--+ +-----------------+
```

```
|tttttttttttttttttttttttttttttt|00|
+------------------------------+--+
```

Instruction Tag Mask : fffffffc hex
Instruction Set Mask : 00000000 hex

DATA ADDRESS DECODER :

```
+----------------------------+--+--+ +------------------+
|3322222222221111111111000000|00|00| |t : tag  bits = 1c|
|1098765432109876543210987654|32|10| |s : set  bits = 00|
+----------------------------+--+--+ |w : word bits = 02|
|tttttttttttttttttttttttttttt|ww|bb| |b : byte bits = 02|
+----------------------------+--+--+ +------------------+
```

Block Address Mask : fffffff0 hex
Sub-block Address Mask : fffffffc hex
Word Address Mask : fffffffc hex
Set Number Mask : 00000000 hex
Sub-block Number Mask : 0000000c hex
Word Number Mask : 0000000c hex
Word Byte Number Mask : 00000003 hex
Block Byte Number Mask : 0000000f hex

Statistics --

Total Number Of Read Requests : 4900537
Total Number Of Write Requests : 2221356
Number Of Read Requests : 4900537
Number Of Write Requests : 2221356
Number Of Read Cancels : 378
Number Of Write Cancels : 0
Number Of Read Hits : 1848517
Number Of Write Hits : 616829
Number Of Transfer Stalls : 0

Total Hits : 1815494
Partial Hits : 33023
Total Misses : 65937
Partial Misses : 2986083
Maximum Write Hits : 569824

Number Of Prefetch Requests : 2076607

Number Of Invalid Predictions	: 2790907
Wrap-Around From Left	: 11502
Wrap-Around From Right	: 134
Prediction in the Same Block	: 2779271
Maximum Pending Prefetches	: 265967
Global Read Hit Ratio	: 0.37720704
Global Read Miss Ratio	: 0.62279296
Global Write Hit Ratio	: 0.27768129
Global Write Miss Ratio	: 0.72231871
Local Read Hit Ratio	: 0.37720704
Local Read Miss Ratio	: 0.62279296
Local Write Hit Ratio	: 0.27768129
Local Write Miss Ratio	: 0.72231871
Block Buffer Read Hits	: 2589
Block Buffer Write Hits	: 4
Block Buffer Read Hit Ratio	: 0.00052831
Block Buffer Write Hit Ratio	: 0.00000180

END OF FILE [iPRC_64k/PRC_dump.00099] -----------------------------------

APPENDIX G. AN EXAMPLE OUTPUT FILE FOR THE BUFFER MODULE

```
+-------------------------------------------------------------------------+
| Module Title   : Buffer1                                                |
| Module ID      : 2                                                      |
| Configuration  : L1_64k          Sat May  2 04:23:58 1998               |
+-------------------------------------------------------------------------+

System Clock : 0071595011 -------------------------------------------

Operating Parameters -----------------------------------------------

Read  Buffer Size              : 8
Write Buffer Size              : 4
Write Buffer Block Size        : 16
Enforce Priorities             : Yes
Remove Read  Duplicates        : Yes
Remove Write Duplicates        : Yes
Search Read  Buffer            : Yes
Search Write Buffer            : Yes

Read Buffer Contents -----------------------------------------------

+------------------------------------------+
| READ  BUFFER [EMPTY]  : 0/8              |
| Access In Progress    : No               |
| # Pushes Attempted    : 443534           |
| # Pushes Granted      : 443534           |
| # Pushes Rejected     : 0                |
+------------------------------------------+

Write Buffer Contents ----------------------------------------------

+------------------------------------------+
| WRITE BUFFER [EMPTY]  : 0/4              |
| Access In Progress    : No               |
| # Pushes Attempted    : 2221822          |
| # Pushes Granted      : 2221822          |
| # Pushes Rejected     : 0                |
+------------------------------------------+

Statistics ---------------------------------------------------------
```

Total Number Of Read Requests	: 4901106
Total Number Of Write Requests	: 2221822
Number Of Read Requests	: 443534
Number Of Write Requests	: 2221822

READ BUFFER :

Number of Requests Slipped	: 0
Number of Requests Dropped	: 0
Total Number of Matches	: 0
Number of Matches (Low-High)	: 0
Number of Matches (High-Low)	: 0
Instruction Address Matches	: 0
Victim Block Matches	: 0
Total Write Hits	: 0
Partial Write Hits	: 0

WRITE BUFFER :

Number of Inclusive Merges	: 0
Number of Adjacent Merges	: 735990
Total Number of Matches	: 0
Number of Matches (Low-High)	: 0
Number of Matches (High-Low)	: 0
Total Read Hits	: 0
Partial Read Hits	: 0

END OF FILE [L1_64k/Buffer1_dump.00099] ----------------------------------

APPENDIX H. AN EXAMPLE OUTPUT FILE FOR THE MAIN MEMORY MODULE

```
+-----------------------------------------------------------------+
| Module Title   : MainMemory                                     |
| Module ID      : 3                                              |
| Configuration  : L1_64k            Sat May  2 04:23:59 1998     |
+-----------------------------------------------------------------+

System Clock : 0071595011 ------------------------------------------

Operating Parameters -----------------------------------------------

Memory Access   Time        : 5
Memory Transfer Time        : 1

Statistics ---------------------------------------------------------

Number Of Read  Requests          : 443534
Number Of Write Requests          : 1484334
Number Of Read  Cancels           : 0
Number Of Write Cancels           : 0

Total Number Of Cycles            : 11372060
Number Of Idle  Cycles            : 60222951
Number Of Read  Cycles            : 3548272   [31.20 %]
Number Of Write Cycles            : 7823788   [68.80 %]

Total Memory Utilization          : 0.15883873
Memory Read  Utilization          : 0.04956033
Memory Write Utilization          : 0.10927840

Average Read  Service Time        : 8.00000000
Average Write Service Time        : 5.27090788
Global  Read  Service Time        : 0.72397375
Global  Write Service Time        : 3.52133870

END OF FILE [L1_64k/MainMemory_dump.00099] ---------------------------
```

LIST OF REFERENCES

1. Patterson, D. A. and J. L Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1996.

2. Heuring, V.P, and H.F. Jordan, *Computer Systems Design and Architecture*, Addison Wesley Longman, Inc. Menlo Park, CA, 1997.

3. Przybylski, Steven, A., *Cache and Memory Hierarchy Design: A Performance Directed Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

4. Handy, J., *The Cache Memory Book*, Academic Press Inc., San Diego, CA, 1993.

5. Altmisdort, N., "Development of a New Prediction Algorithm and a Simulator for the Predictive Read Cache (PRC)," Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1996.

6. Fouts, D.J. and A. B. Billingsley, "Predictive Read Caches: An alternative to On-Chip Second-Level Cache Memories," *Journal of Microelectronic Systems Integration*, vol. 2, no. 2, 1994.

7. Miller, R.W., "Simulation and Analysis of Predictive Read Cache Performance," Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1992.

8. Grimsrud, K., J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "BACH: A Hardware Monitor for Tracing Microprocessor-Based Systems," *Microprocessors and Microsystems*, vol. 17 no. 8, October 1994.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5002

3. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

4. Professor Douglas J. Fouts, Code EC/Fs 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

5. Professor Frederick W. Terman, Code EC/Tz 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

6. LT Kathryn Christensen 1
   4534 Calle de Vida
   San Diego, CA 92124