NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

SIMULATION AND ANALYSIS OF PREDICTIVE READ CACHE PERFORMANCE

by

Robert W. Miller

June, 1995

Thesis Advisor: Douglas J. Fouts

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	Master's Thesis

4. TITLE AND SUBTITLE SIMULATION AND ANALYSIS OF PREDICTIVE READ CACHE PERFORMANCE

5. FUNDING NUMBERS

6. AUTHOR(S) Miller, Robert W.

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)*

Efforts to speed up the memory hierarchy have failed to keep up with the rapid increase in microprocessor performance. The use of first-level and second-level caches has become common in an effort to minimize this speed discrepancy. One potential method to overcome the speed problem, while using much less hardware than a second-level cache, is the predictive read cache. This thesis continues previous efforts in designing and optimizing the predictive read cache. It develops a method to simulate the performance of a memory hierarchy containing a predictive read cache and uses these simulations to determine the most effective architecture of the cache. Using trace data from an Intel 486 processor running the SPEC benchmarks, the simulations demonstrate that a small predictive read cache can give a performance improvement equivalent to a much larger second-level cache. This makes the predictive read cache ideal for systems that are power or chip area limited.

14. SUBJECT TERMS Cache, Predictive, Memory			15. NUMBER OF PAGES 56
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT 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.

SIMULATION AND ANALYSIS OF PREDICTIVE READ CACHE PERFORMANCE

Robert W. Miller
Lieutenant Commander, United States Navy
B.S.E.E., United States Naval Academy, 1980

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1995

Author: Robert W. Miller
Robert W. Miller

Approved by: Douglas J. Fouts
Douglas J. Fouts, Thesis Advisor

Shridhar B. Shukla, Second Reader

Michael A. Morgan
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Efforts to speed up the memory hierarchy have failed to keep up with the rapid increase in microprocessor performance. The use of first-level and second-level caches has become common in an effort to minimize this speed discrepancy. One potential method to overcome the speed problem, while using much less hardware than a second-level cache, is the predictive read cache. This thesis continues previous efforts in designing and optimizing the predictive read cache. It develops a method to simulate the performance of a memory hierarchy containing a predictive read cache and uses these simulations to determine the most effective architecture of the cache. Using trace data from an Intel 486 processor running the SPEC benchmarks, the simulations demonstrate that a small predictive read cache can give a performance improvement equivalent to a much larger second-level cache. This makes the predictive read cache ideal for systems that are power or chip area limited.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. THE NEED FOR CACHE MEMORIES

Microprocessor design improvements have resulted in dramatic speed increases over the last few years. The performance of the latest high-end microprocessors has increased at a rate of 54 % per year over the last ten years [Ref. 1]. However, the speed of the dynamic random access memory (DRAM) required for all modern computers has failed to keep up with the increases in microprocessor speed, only improving by a factor of three (180 ns access time vs. 60 ns access time [Ref. 2]).

To get around the problem of slow main memory inhibiting the performance of the microprocessor, designers have gone to a hierarchical memory design where a smaller and faster memory (a cache) is placed between the microprocessor and the main memory. This cache memory allows most memory references to be handled by the cache at a high speed with only a few references requiring slower main memory access. Use of a cache memory has become so critical to the operation of modern microprocessors that it has become mandatory to incorporate one directly on the same chip as the microprocessor (for example, the 8 Kbyte cache on the Intel 486 microprocessor [Ref. 3]). An on-chip cache is limited in size due to silicon area restrictions and the requirement that its data be available in one processor clock cycle. Because of the importance of this on-chip cache, numerous studies have been done on determining its optimum configuration and most practical performance improvements have already been implemented.

Initially, a single cache provided the improvement in average memory access time needed to allow the nominal microprocessor to operate efficiently. However, as the speed and memory bandwidth requirements of later microprocessors increased, designers soon had to add off-chip second-level caches. A second-level cache could be larger and slower than the first-level cache since there are no chip area limitations, cache access time is not directly tied to microprocessor clock rate, and any improvement in access time over that

of main memory is beneficial. Although a second-level cache provides a performance enhancement, use of an off-chip cache has several disadvantages. The most important is that it is connected to the first-level cache through the microprocessor package's input/output pins and therefore has a limited width datapath that reduces the rate at which data can be passed from the off-chip cache to the microprocessor. Additionally, since the second-level cache is external to the microprocessor package, it requires additional components and control circuitry which adds to system complexity and cost.

The next logical step in improving memory access time is to move the second-level cache into the same package as the microprocessor. The Intel Corporation's P6 microprocessor will have separate 8 Kbyte first-level data and instruction caches on the microprocessor chip and a 256 Kbyte second level cache on a separate silicon chip inside the same package [Ref. 4]. This design allows a wider datapath between caches and allows the designer to better optimize the memory hierarchy. The Digital Equipment Corporation's Alpha 21164 goes even further with an integrated 96 Kbyte second-level cache on the same chip as separate 8 Kbyte data and instruction first-level caches. However, this implementation requires over nine million transistors on the chip.[Ref. 5]

As can be seen, the use of large second-level caches is very expensive and complicates the system design. A very different method to gain the improved memory access times without the expense of a second level cache has been proposed by Fouts and Billingsly [Ref. 6]. Their solution is to us an on-chip predictive read cache (PRC) instead of a second-level cache. By monitoring the trend in memory accesses, the PRC prefetches data from main memory to have it available when it is needed by a first-level cache miss.

B. CACHE MEMORIES

Before describing the PRC, it is necessary to have a basic understanding of cache memories and how cache performance is normally measured. As mentioned earlier, a cache consists of a small high-speed memory located between the microprocessor and main memory. Since it is smaller than main memory, it can only hold a subset of the data

contained in main memory. However, due to the spatial and temporal locality of memory references exhibited by most programs, the cache can contain a significant proportion of the microprocessor memory accesses. This fact is the primary reason caches improve the operation and speed of a microprocessor system [Ref. 1].

Common measures of cache performance are the hit and miss ratios. The hit ratio is defined as the fraction of memory accesses that were found in the cache memory while the miss ratio is 1 - hit ratio [Ref. 1]. Most modern day first-level caches have a hit ratio in the low 90% range and most cache optimization has been concerned with improving the hit rate since it is the most easily measured parameter during simulations. However, another significant aspect of cache performance is the miss penalty; that time required to get the data when it is not in the cache. The miss penalty is significantly affected by the downstream memory hierarchy and a high miss penalty can offset the benefits of a high cache hit ratio [Ref. 2]. Combining the miss penalty with the hit rate yields the average memory access time [t_{access} = HitRate x t_{cache} + MissRate x t_{miss}] where t_{cache} = cache access time and t_{miss} = miss penalty. Therefore, the average memory access time serves to summarize the performance of the entire memory hierarchy and more accurately reflects the impact of cache and memory hierarchy design decisions on microprocessor system performance.

C. PREDICTIVE READ CACHE

The predictive read cache (PRC) is fully described by Fouts [Ref. 6]. It is a cache memory, organized in any of the standard methods, with the difference that the data retrieved into the PRC is based on a predictive algorithm. It is by virtue of this predictive algorithm that a small PRC could potentially outperform a larger second-level cache.

One of the first observations made by Fouts [Ref. 6] is that the PRC is best suited for only data references since there already exist several methods to improve the retrieval of instructions (i.e., prefetch queues). Therefore, the best location for the PRC would be

between a first-level data cache (separate from the instruction cache) and main memory. This allows the PRC to be optimized for the distribution of data references in memory.

The designers of the PRC noted that in a typical multitasking environment, groups of data references exhibit a strong spatial locality with temporal interleaving. This requires the PRC to maintain several different prediction traces so that task switches will not invalidate the prediction efforts. The prediction algorithm itself is very simple: the difference between the next (predicted) address and the current reference address should be the same as the difference between the current reference address and the last reference address fetched. This design requires a standard cache memory to hold the data and associated address tags, additional storage to maintain the previous PRC miss address, the address differences for each cache line, and the hardware necessary to implement the predictive algorithm. But, since only a very small PRC is required to provide a significant performance improvement, one could be implemented without using too much valuable real estate.

To better understand the PRC, a simple example is required. First, assume address 000100 is a first-level cache miss. At the same time this address is sent to the read buffer for main memory access, the contents of the PRC are checked. Since this is the first access to the PRC it will be a miss and the main memory read will be required to satisfy the cache request. Next, assume address 000110 also misses the cache. Again, a main memory read is started and the PRC is searched. Since this data is also not in the PRC, the memory read continues and data is retrieved to satisfy the cache request. However, the PRC will now compute the next predicted access [(000110) + (000110 - 000100) = 000120] and will initiate a memory read for that data. If, as expected, the next cache miss is for address 000120, the data will be available in the PRC and no main memory read will be required. This will significantly improve the memory access time. Additionally, if there is an access for address 000120, the PRC will fetch the data at address 000130 in anticipation of the next request from the cache. Then, if due to a task switch, the next

request is for another address, the cycle described above will repeat for the new sequence of addresses with the predicted data being stored in a different PRC location. This allows the PRC to have data available for both predicted addresses when they are needed to satisfy subsequent cache misses. This prediction cycle is repeated for all first-level cache misses.

D. THE NEED FOR SACS2

In order to determine the effectiveness of a PRC and to properly optimize its parameters, a cache and memory hierarchy simulator that provides more than just hit and miss percentages is required. The average memory access time is the critical information that properly accounts for the effects of the PRC. The optimum PRC parameters would be indicated by the minimum average memory access time. Comparing access times between configurations with and without a second-level cache would reveal the comparative effectiveness of a PRC to these other options.

Smith [Ref. 7] developed a cache simulation program (SACS - Still Another Cache Simulator) that provides exactly the information necessary to optimize a cache memory for minimum access time. It was designed to use address traces in ASCII format and simulates a single-level cache interacting with main memory. By building on that foundation, this thesis documents the programming changes required to incorporate a PRC into the simulation memory hierarchy and the results of exhaustive testing using the modified simulator (SACS2).

II. DESIGN OF SACS2

A. INTRODUCTION TO SACS2

SACS2 is a cache and memory hierarchy simulation program written in C. It is a modification and enhancement to the original SACS written by Smith [Ref. 7] and is designed to aid in the analysis of the effects of adding a PRC to a memory hierarchy containing a first-level cache. To implement this design goal required the addition of functions to model the PRC and modifications to other program functions to incorporate the PRC. Additionally, to allow comparison testing of a memory hierarchy built with both a first-level and second-level cache, another modified program (SACS21) was written. This program uses SACS2 as its basis and incorporates the changes necessary to model a non-predictive second-level cache.

To provide for testing of the various cache and PRC design options, SACS2 gives the user the ability to vary numerous design parameters. This allows determination of the optimal first-level cache and PRC configuration by the running of simulations with different combinations of design parameter values. The user can then compare the resulting average memory access times and choose the combination that yields the best results.

B. SACS2 INPUT PARAMETERS

The various adjustable SACS2 input parameters are shown in Tables 1 and 2. These parameters are specified in a *sacs.ini* file so that they can be easily modified for the different simulation runs. This is a modification from the original SACS program that required the user to adjust simulation parameters using command line switches [Ref. 7]. The parameters listed in Table 1 are unchanged from their usage in the original SACS and are fully discussed by Smith [Ref. 7] and therefore will not be separately discussed here. However, the impact of changes in these parameters during the simulation runs will be discussed in Chapter III. The parameters listed in Table 2 were added in SACS2 to allow for proper modeling of the PRC and will be discussed further in this section.

Cache Size	Search Block Buffer
Block Size	Update Read Buffer
SubBlock Size	Remove Read Duplicates
Cache Associativity	Read Buffer Size
Word Size	Write Buffer Size
Read Cache Access Time	Cache Block Replacement Policy
Read Cache Hit Time	Cache Write Policy
Read Cache Miss Time	Cache Write Miss Policy
Write Cache Access Time	Cache Read Forward
Write Cache Hit Time	CPU Waits for Cache Writes
Write Cache Miss Time	Remove Write Duplicates
Memory Access Time	Read Priority
Memory Transfer Time	Read For Write Allocate Priority
Buffer Cache Access Time	Write Dirty Block Priority

Table 1 Input Parameters From Original SACS

Use PRC	PRC Block Replacement Policy
PRC Size	PRC Write Policy
PRC Associativity	Use PRC on Write Miss
PRC Cache Access Time	PRC Read Priority
Buffer PRC Access Time	PRC Slipped Read Priority
Max PRC Size in Buffer	Drop PRC on Second Miss

Table 2 Input Parameters Associated with PRC

1. Use PRC, PRC Size, and PRC Associativity

Use PRC is provided to allow simulation runs without a PRC so that data for a memory hierarchy with only a first-level cache could be obtained for comparison purposes. *PRC Size* is the size in bytes of the PRC being modeled. *PRC Associativity* allows simulating any degree of PRC associativity from direct mapped to fully associative. The model used in SACS2 assumes that the PRC Block Size and PRC Sub Block Size are the same as the Cache Block Size and Cache Sub Block Size respectively. It also assumes that the datapath between the PRC and cache is one block wide allowing an entire cache block to be transferred at once from the PRC.

2. PRC Access Times

The *PRC Cache Access Time* is used to model the time it takes for data in the PRC to be transferred to the cache on a PRC hit. It models the time required to check the address tags and valid bits along with the time required to actually transfer the data on a PRC hit. *Buffer PRC Access Time* is used to model the time it takes for the Block Buffer to access the PRC and load it with incoming data once the data has been read from memory.

3. PRC Read Parameters

There are four parameters directly associated with the handling of PRC read requests in the read and block buffers. *PRC Read Priority* is the priority associated with the predictive read being done by the PRC. It is used by the read buffer to determine the sequencing of read requests to main memory. It would normally be set to a lower value than the cache miss read priority to ensure that cache misses are handled first. *Max PRC Size in Buffer* is another parameter used to ensure that cache miss reads receive priority. Normally in the memory model, any read that has started access to memory gets the highest priority and will continue until complete. However, this could force a cache miss read request to be delayed significantly if it enters the read buffer just after a PRC read has started. To prevent this, the simulator has been designed to allow the cache read request to push a PRC read request that is not too far along off the top of the buffer so the

cache read request can immediately start. The *Max PRC Size in Buffer* parameter determines the maximum portion of the original PRC read that can be remaining and still allow the PRC read to continue. This parameter is critical because, although a PRC read should not hold up a cache read, continually stopping almost complete PRC reads would cause these incomplete reads to fill up the read buffer and no PRC read requests will ever get processed. If a PRC read is pushed off the top of the buffer, its priority is assigned the *Slipped PRC Read Priority*. This allows the designer to give these read requests a lower priority since, if they have slipped once, they are more likely to be superseded by a cache read for the same information and/or a new PRC read for the next predicted address. *Drop PRC On Second Miss* allows the designer to specify whether PRC read requests that have been bumped off the top of the read buffer twice should be canceled or left in the buffer.

4. PRC Block Replacement Policy and PRC Write Policy

The *PRC Block Replacement Policy* controls how locations for the storage of new PRC read request data will be determined. It can be set to Least Recently Used (LRU), Random, or First In First Out (FIFO) allowing simulation for any of these three common cache replacement policies. The *PRC Write Policy* dictates what happens in the PRC on a cache write to memory. If *PRC Write Policy* is *Write Through,* the contents of the PRC for the write address are updated along with main memory. If *PRC Write Policy* is *Write Around,* the contents of the PRC for the write address are simply invalidated.

5. Use PRC On Write Miss

The setting of *Use PRC On Write Miss* determines whether the PRC will be searched when there is a memory read caused by a write allocate fill of a block in the first-level cache. If *Use PRC On Write Miss* is *Yes* the PRC will be searched and a new prediction trace started based on the address of this read request. If *Use PRC On Write Miss* is *No* the PRC is ignored on these write allocate reads and the cache read request goes directly to the read buffer.

C. SACS2 OPERATIONAL DETAILS

SACS2 operates in essentially the same manner as the original SACS [Ref. 7]. There is a main event loop where simulation time is incremented. All other functions simulating portions of the memory hierarchy are called from the main event loop. The main event loop ensures that all actions that can be done for a given simulation time are completed before incrementing the simulation time. Since there are many similarities between SACS2 and SACS, only the new functions and the significantly modified functions are discussed here.

1. Program Initialization

SACS2 uses a *sacs.ini* file to store all the user-definable parameters for the simulation. If necessary, a file with a name other than *sacs.ini* can be used by specifying its name on the command line. When SACS2 is run, the function *LoadArguements* parses the *sacs.ini* file to retrieve the initialization values for the user-definable parameters and stores these values for use. The original SACS also required that the address traces used in the simulation be formatted as ASCII text [Ref. 7]. To allow use of available actual address traces, SACS2 has been modified to use binary traces in the BYUTR format created by the BACH trace generation tools [Ref. 8]. These readily available traces generated by Intel 486 and Sparc microprocessors running the SPEC benchmarks include all of the key parameters necessary for the SACS2 simulation (address, read/write, instruction/data, time). The length of these traces ensures a more accurate analysis.

2. PRC Functions

a. PRCRead

PRCRead is called whenever there is a first-level cache read miss or a first-level cache write allocate read (if *UsePRCOnWriteMiss* = Yes) on a write miss. It calls *IsPRCRequestAHit* to determine if the read request is in the PRC. Either *PRCReadHit* or *PRCReadMiss* are then called based on the results of *IsPRCRequestAHit*.

b. IsPRCRequestAHit

IsPRCRequestAHit determines whether the request is in the PRC by searching all PRC tags for the request address and, if the address is found, checking the associated valid bits. If the request is a hit, the *CurrentDeltaAddress* is set to the *PRCDeltaAddress* that has been stored with the hit address. *CurrentDeltaAddress* is the information that the PRC uses to predict the address of the next requested data, and for a PRC hit, will be the same as that used to retrieve the data that was found by the PRC hit . If the request is a miss, the *CurrentDeltaAddress* is instead set to the difference between the address of the current request and the address of the last PRC miss in accordance with the PRC algorithm [Ref. 6].

c. PRCReadHit

PRCReadHit first calls *AccessPRC* to determine if the cache PRC access time has elapsed. If it has not, no action is taken during the current simulation time. Once the cache PRC access time has elapsed, *UpdateCacheFromPRC* is called to update the cache with the data found in the PRC. *PRCReadHit* then determines the next address to retrieve based on the PRC algorithm *(PredictedAddress = CurrentAddress + CurrentDeltaAddress).* Before placing the *PredictedAddress* in the read buffer using *PRCAddToReadBuffer,* the *PredictedAddress* is checked to make sure it is not being generated by a wraparound through zero (if *CurrentDeltaAddress* is negative and larger than *CurrentAddress*) and also that the *PredictedAddress* will not retrieve the same PRC block that includes the *CurrentAddress* (as may occur if the *CurrentDeltaAddress* is less than the block size). These checks are done so that unnecessary PRC reads of memory will be eliminated to prevent filling up the read buffer and slowing down other memory accesses by using memory bandwidth.

d. PRCReadMiss

PRCReadMiss also calculates the *PredictedAddress* by adding *CurrentDeltaAddress* to the *CurrentAddress.* As was done in *PRCReadHit*, it checks for zero wraparound and that the *PredictedAddress* is not in the same block as the

CurrentAddress. In this case, if the *PredictedAddress* was in the same block as the *CurrentAddress*, the PRC would just be adding a request to the read buffer for the same data that the cache read request is already retrieving.

e. PRCWrite

PRCWrite is called whenever the first-level cache has to perform a write to main memory. It checks the PRC for the presence of the data being overwritten by the memory write and, depending on the value of *PRCWritePolicy*, either invalidates the data (for *PRCWriteAround*) or copies the new data into the PRC (for *PRCWriteThrough*).

f. AccessPRC

AccessPRC is used to simulate the cache PRC access time associated with a read hit. It operates similarly to *AccessCache* and ensures that the delays associated with the PRC tag search and cache transfer are correctly accounted for.

g. SelectPRCBlockVictim

SelectPRCBlockVictim is called to search the PRC and determine which block will store the new line of data for the predicted address being requested from main memory. If the read request is being generated by a PRC hit and a fully associative PRC is being used, *SelectBlockVictim* will choose the block where the PRC hit was generated. This maximizes the efficiency of the PRC since it minimizes the overwriting of other potentially useful blocks. If the PRC is not fully associative, the block to be used will be determined by applying a LRU, Random, or FIFO algorithm depending on the setting of *PRCBlockReplacementPolicy. SelectPRCBlockVictim* also takes care of storing the address tag in the selected block.

h. SetPRCValidBits

SetPRCValidBits is called whenever it is necessary to invalidate a block in the PRC such as on a cache write when *PRCWritePolicy* is *PRCWriteAround*. It loops through the sub-blocks in the selected block and clears their valid bits.

i. PRCAddToReadBuffer

PRCAddToReadBuffer is responsible for putting the read request generated by the PRC into the read buffer. It first checks if a PRC read request for the same block is already in the buffer and, if so, does nothing more since the data needed is already being retrieved. Next, it checks if a PRC read request for a different address but for the same PRC block is in the read buffer. If this is the case, it eliminates the older read request since the data being retrieved would be immediately overwritten by the data being retrieved by the new request. If neither one of these conditions is present, it stores the *CurrentDeltaAddress* in the PRC block being updated and then adds the read request to the read buffer.

j. UpdateCacheFromPRC

UpdateCacheFromPRC is called when there is a PRC hit and is responsible for the updating of the cache valid and dirty bits to indicate that the data requested has been transferred from the PRC to the cache. It is called by *PRCReadHit* after the Cache PRC Access time has elapsed.

3. Cache Functions

a. ReadMiss

ReadMiss has been updated from the original SACS to account for the use of a PRC. Once *ReadMiss* has selected the cache block for the miss request, it calls *PRCRead* to check if the data is available in the PRC. If *PRCRead* returns with a PRC hit, *ReadMiss* takes no further action. If *PRCRead* returns with a PRC miss, *ReadMiss* must add the read request to the read buffer. Before doing so, it checks to see if a PRC read request is currently in progress which could slow down the retrieval of the data necessary to satisfy the cache miss. If such a PRC read request is in progress and the number of bytes left to retrieve is greater than the value of *MaxPRCSizeInBuffer*, the PRC read request is bumped off the read buffer and reset to a lower priority. This ensures the cache read request being added will immediately start to access memory. Next *ReadMiss* checks to see if there is a PRC read request currently in the read buffer that is retrieving the same

data as the current cache read request. If there is such a request, it is cancelled as its retrieval would be redundant and waste memory bandwidth. Finally, *ReadMiss* adds the cache read to the read buffer using *AddToReadBuffer*.

b. WriteHit and WriteMiss

WriteHit has been updated to call *PRCWrite* so that the correct action is taken by the PRC on all writes to memory. *WriteMiss* also calls *PRCWrite* but in addition, if a write allocate cache strategy is in use and *UsePRCOnWriteMiss* = *Yes*, it calls *PRCRead* to check for the data when it performs the read to fill the rest of the block being written to.

4. Buffer Functions

Several of the buffer functions have been modified to account for the use of the PRC. By design, the source of the memory read (cache miss or PRC miss) is stored with the memory request so that the correct location (cache or PRC) is updated when the data becomes available from memory. Therefore, functions (such as splice) that search the buffers for duplicate data had to be modified to only look for those requests that were generated by the same source as the one currently adding the request to the buffer. This ensures that the cache and PRC are correctly updated by their associated read requests.

5. Memory Functions

a. ContinueMemoryReads

ContinueMemoryReads is the function that simulates the transfer of words from memory after the initial word has been retrieved into the block buffer. It is also responsible for ensuring that the correct internal variables are set so that the cache or PRC (as appropriate) will be updated once the entire request has been read from memory and is stored in the block buffer. Therefore, it had to be modified to account for the presence of the PRC.

b. UpdatePRC

UpdatePRC is a new function that simulates the transfer of data from the block buffer to the PRC once a PRC memory read is complete. It is structured similarly

to *UpdateCache* and is responsible for updating the PRC valid bits for the block being transferred once the buffer PRC access time has elapsed. It additionally removes the data from the block buffer so the next request in the read buffer can commence.

D. MODIFICATIONS TO SIMULATE A SECOND-LEVEL CACHE

To allow comparison of the performance of a PRC with that of an on-chip second-level cache, another set of modifications had to be made to the SACS program to model the second-level cache. These modifications to the cache, PRC, buffer, and memory functions of SACS2 resulted in SACS21.

1. Cache Functions

a. ReadMiss

ReadMiss was modified so that all read requests generated by first-level cache misses went through the second-level cache and no read requests could be added to the read buffer directly by the first-level cache. This design could be used since all first-level cache misses would either be satisfied by the second-level cache or by read requests generated by second-level cache misses. If a second-level cache miss occurred, the first-level cache would be updated at the same time as the second-level cache.

b. AddToReadBuffer

AddToReadBuffer was modified to be used by the second-level cache for additions to the read buffer. To provide the ability to update both caches when the read was complete, the data structures used for read requests were modified to include both of the associated cache blocks.

2. PRC Functions

All PRC functions were modified so that the PRC would act like a second-level cache and not a predictive cache. This required elimination of all references to delta addresses and all calculations of predicted addresses. Additionally, the *PRCWriteHit* and *PRCWriteMiss* functions were modified to be similar to the cache *WriteHit* and *WriteMiss* functions so that the different write and write miss policies available for caches could also be simulated in the second-level cache.

3. Buffer Functions

Buffer functions were modified to eliminate the difference between cache reads and PRC read requests since now all requests would come from the PRC (acting as a second-level cache) and would be used to update both caches. This required a modified read request data structure and a modification to the search functions used for the scoreboarding protocol.

4. Memory Functions

The only change to the memory functions required was to the *UpdatePRC* function. It was modified so that, upon the completion of a memory read, both the PRC (second-level cache) and the first-level cache would be updated with the new data. This required adding the cache update loop from *UpdateCache* into the *UpdatePRC* function.

III. SIMULATION RESULTS AND ANALYSIS

A. SIMULATION ASSUMPTIONS

The SACS2 simulations used to conduct the testing detailed in this thesis were based on certain assumptions that have an impact on the results obtained. Some of these assumptions were incorporated into the design of SACS2 while others were used to set the value of constant parameters.

1. Assumptions Built Into SACS2 Model

In designing the cache and PRC modeling functions used to perform the simulations detailed in this thesis, certain design decisions were made and built into the SACS2 program. Fouts [Ref. 6] discussed that a PRC should only be used to predict data references since there were already alternate methods to speed up the retrieval of instruction references. Therefore, SACS2 analyzes the address trace data and only operates on the data references contained in it. Because of this, the program models separate instruction and data caches with all instruction cache details ignored. This assumption does not affect the general trend of data cache and PRC behavior and the results obtained should be valid.

The method of modeling the PRC and its interaction with the first-level cache assumes that the PRC is located on-chip between the first-level cache and the buffers that interface with main memory. Additionally, the PRC and first-level cache are connected by enough data lines to pass one complete cache block at a time. These assumptions are valid based on the small size of the PRC and the current trends in microprocessor design.

According to the PRC design parameters of Fouts [Ref. 6], the model assumes that cache misses go to the read buffer and PRC simultaneously. If the data is then found in the PRC, the associated memory read is canceled. This method is used so that cache misses that are also PRC misses do not take longer than they would without a PRC. This method is achievable in current microprocessor design and ensures that the presence of a PRC does not slow the system.

Since a PRC read for information that is already in the first-level cache would be redundant and a waste of memory bandwidth, the model assumes that a cache miss read request that goes to the read buffer can check the buffer for the presence of PRC read requests for the same address and cancel those requests. This type of scoreboarding is consistent with current microprocessor design and ensures that the limited memory bandwidth is put to the best use.

Two assumptions are made in designing SACS21 where the PRC is simulating a second-level cache. In this case, all first-level cache misses go to the second-level cache and any memory read request must wait until after it has been determined that a miss has occurred in both caches. Additionally, since both caches are assumed to be collocated on the chip, all data arriving from memory read requests is made available to both caches on arrival. This allows the first-level cache to satisfy its memory request as soon as the needed data is read from memory and allows both caches to be updated with the new data block simultaneously.

B. CONSTANT CACHE MODEL PARAMETERS USED

Since there are a large number of user definable parameters in the SACS2 model, assumptions had to be made to decide which parameters should be constant and what their values should be. The values of the parameters that remain constant for all simulations are summarized in Table 3 and explained below.

1. First-Level Cache Parameters

The address traces used in the simulations for this thesis were taken from an Intel 486 microprocessor running a UNIX operating system. Because of this, most of the first-level cache parameters were set to match those on the 486. A 4-way set associative 8192 byte cache with 16 byte blocks and 4 byte words was chosen as the most accurate model for the 486. However, to model a cache closer to the current state of the art, a true LRU cache block replacement algorithm and a write back write policy rather than write through were used. Additionally, a write allocate miss strategy was used so that all writes to the first-level cache would generate a complete valid block of data in the cache.

First-Level Cache Parameters	Buffer Parameters
Size = 8192 bytes	Read Buffer Size = 8
Block Size = 16 bytes	Write Buffer Size = 4
Sub Block Size = 4 bytes	Access in Progress Priority = 0
Word Size = 4 bytes	Cache Read Miss Priority = 1
Associativity = 4 way set associative	Write Priority = 2
Block Replacement Policy = LRU	Read For Write Allocate Priority = 3
Write Policy = Write Back	Write Dirty Cache Sub Blocks Priority = 4
Write Miss Policy = Write Allocate	PRC Predictive Read Priority = 5
Cache Miss Actions	Slipped PRC Read Priority = 6
Read Forward = Yes	**Access Times**
CPU Wait for Writes = No	Read Cache Access Time = 1
Search Block Buffer = Yes	Write Cache Access Time = 1
Update Read Buffer = Yes	Cache Hit/Miss Access Times = 0
Remove Read Duplicates = Yes	Cache PRC Access Time = 1
Remove Write Duplicates = Yes	Memory Access Time = 5
	Memory Transfer Time = 1
	Buffer Cache/PRC Access Time = 1

Table 3 Constant Simulation Model Parameters

2. Cache Miss Actions

The cache miss actions detailed in Table 3 were chosen to accurately model current cache design practices. Read Forward ensures that every first-level cache miss will generate a read request that will completely fill the cache block. CPU Wait for Writes models the use of a write buffer where any writes to memory can be handled in parallel with more cache accesses. Search Block Buffer provides the ability for a read request generated by the first-level cache to be satisfied if the data needed is currently in

the block buffer due to a previous read request. This can significantly improve performance by reducing the time it takes to retrieve the cache miss data. Update Read Buffer models the ability of a memory write request to check the read buffer for pending read requests and remove any bytes stored in the cache by the write. This reduces the size of the read request and prevents overwriting of the new data in the cache with older data being read from main memory. Remove Read/Write Duplicates ensures that only one request for each cache block is active at a time and, if a new request is generated for data that is already in the buffers, it is merged with the pending requests.

3. Buffer Parameters

The read buffer size was set at eight requests based on the results of preliminary testing of SACS2. With the addition of a PRC there are many read requests generated and, with a small read buffer, requests generated by cache misses could be blocked by a full buffer. With the larger buffer, these requests can enter the buffer and be included in the ordering of memory accesses by priority. The write buffer size was set at four requests to be consistent with current cache design since the PRC does not significantly affect the number of writes to memory. The buffer priorities were chosen to be consistent with the design parameters of a cache and PRC memory hierarchy. Satisfying a current cache miss needs to occur as soon as possible so it was given the highest priority. Predictive read requests were given the lowest priorities so that they would have minimal impact on other cache operations.

4. Access Times

The various timing parameters used in the simulations were chosen to most accurately reflect current microprocessor and memory design. These parameters are given as multiples of the system clock cycle so that the resulting average access times are independent of the actual clock speed. A time of one clock cycle was chosen for the cache access time for both reads and writes. This access time is the time it takes for the first-level cache to search its tags and determine if the request is present in the cache. If the request is present, the cache read/write hit times of zero cycles result in the data request

being satisfied during the current clock period. If the request is a miss, the cache miss times of zero cycles result in the associated request going to the PRC and to main memory during the current clock cycle. If this request to the PRC results in a PRC hit, the cache PRC access time of one cycle will make the data available to the first-level cache on the next clock period. The memory access time of five cycles determines how long after a request is made to memory until the first word of the data is read into the block buffer. This models the current large discrepancy between microprocessor clock rates and memory access times. Once the first word of the request has been read from memory, the following words are then transferred, one every clock period. Finally, the buffer cache/PRC access times of one period model the delay associated with the transfer of data from a full block buffer to the cache or PRC.

C. SIMULATION RESULTS

Numerous simulations were run to test the effects of varying PRC design parameters on the performance of the memory hierarchy. The parameters varied are summarized in Table 4 and the effects of their variation will be discussed below.

PRC Size	PRC Write Policy
PRC Associativity	Use PRC on Write Miss
Max Size in Buffer to Continue Read	Drop PRC on Second Slip
PRC Block Replacement Policy	

Table 4 PRC Design Parameters

The simulations were conducted using address traces from the BACH trace generation system developed at BYU [Ref. 8]. Three traces were used for most of the testing. The first trace was generated during the running of the eqntott portion of the SPEC benchmarks and is approximately 1.2 million data references long. It is referred to below as trace EQNT. Eqntott is a program that converts boolean equations to the equivalent truth tables. In doing so, it performs numerous sorts of a reasonably compact data set. The

second and third traces were both generated during the running of the kenbus benchmark (20 users). The kenbus program is designed to represent Unix/C usage in a research and development environment with twenty concurrent users. It uses Unix shell scripts to exercise the entire computer system and access data from many varied memory locations. The second trace is approximately 2.4 million data references long and is referred to below as trace KEN1. The third trace is approximately 4.7 million data references long and is referred to as KEN2. These traces were chosen as representative of both single user with compact data storage (EQNT) and multi-user (KEN1 and KEN2).

1. Baseline Testing with No PRC

To establish a comparison baseline, simulations were run for all three traces with the PRC disabled. This provided the performance of a first-level cache alone. The results are shown in Table 5. These results show that despite a reasonably high cache hit percentage of over 91%, the cache miss penalty increases the average read access time about 40% over that of a perfect cache where all accesses would take one cycle. The buffer read hit rate indicates the percentage of cache misses that are found in the block buffer.

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	91.8	13.2	1.377169	92.9	49.5	1.934060
KEN1	91.4	6.1	1.423514	94.9	87.3	1.948817
KEN2	91.5	6.0	1.416308	95.2	85.4	1.951912

Table 5 Cache Performance Without a PRC

2. Effects of PRC Size on Performance

The next set of simulations involve the addition of a PRC to the memory hierarchy and observation of its effects on performance. Table 6 shows the performance with a 256 byte 4-way set associative PRC added to the hierarchy. As can be seen, the

PRC hit rate was approximately 20% indicating the PRC correctly predicted the data request coming from the cache one out of five times. The addition of this very small PRC results in a performance speedup ($SpeedUp = \frac{OriginalAccessTime - AccessTimeWithPRC}{OriginalAccessTime}$) of between 4.6% and 4.9% for read accesses. This is a significant improvement for the small amount of additional hardware required to implement the PRC. There was negligible change in the performance of write accesses since the PRC predicts memory reads and is designed to not affect writes. The small changes in the write performance are due to an increased number of buffer hits where the PRC has brought the data into the block buffer just prior to the write.

Next, simulations of larger PRC sizes were run to determine the effect of PRC size on performance. For simulations of 512, 1024, 2048, 4096, 8192 and 16384 byte PRCs, a 4-way set associative PRC was again used to be consistent with current design practices. The results of these simulations are given in Tables 7 through 12. A summary of the simulation results for the various PRC sizes is given in Table 13 while Figure 1 shows the results graphically.

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.4	22.2	1.310371	92.9	55.6	1.934213
KEN1	91.4	6.7	19.4	1.357973	94.9	87.5	1.948792
KEN2	91.6	6.5	20.2	1.349544	95.2	85.9	1.951913

Table 6 Performance Using 256 Byte 4-way Set Associative PRC

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.4	22.6	1.308821	92.9	55.5	1.934439
KEN1	91.4	6.7	19.8	1.356720	94.9	87.5	1.948810
KEN2	91.6	6.5	20.5	1.348318	95.2	85.9	1.951887

Table 7 Performance Using 512 Byte 4-way Set Associative PRC

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.4	23.0	1.307791	92.9	55.5	1.933843
KEN1	91.4	6.7	20.0	1.355939	94.9	87.5	1.948775
KEN2	91.6	6.5	20.8	1.347330	95.2	85.9	1.951905

Table 8 Performance Using 1024 Byte 4-way Set Associative PRC

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.4	23.9	1.306571	92.9	55.5	1.934582
KEN1	91.4	6.7	20.4	1.354370	94.9	87.5	1.948809
KEN2	91.6	6.5	21.3	1.345649	95.2	85.9	1.951912

Table 9 Performance Using 2048 Byte 4-way Set Associative PRC

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.4	23.9	1.304870	92.9	55.5	1.934135
KEN1	91.4	6.8	21.1	1.351975	94.9	87.5	1.948786
KEN2	91.6	6.5	22.0	1.342989	95.2	85.9	1.951910

Table 10 Performance Using 4096 Byte 4-way Set Associative PRC

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.3	25.0	1.301063	92.9	55.5	1.934111
KEN1	91.4	6.8	22.2	1.348034	94.9	87.5	1.948784
KEN2	91.6	6.6	23.3	1.338538	95.2	85.9	1.951904

Table 11 Performance Using 8192 Byte 4-way Set Associative PRC

Trace	Cache Read Hit Rate %	Buffer Read Hit Rate %	PRC Read Hit Rate %	Read Average Access Time	Cache Write Hit Rate %	Buffer Write Hit Rate %	Write Average Access Time
EQNT	92.4	8.3	27.3	1.292856	92.9	55.5	1.934095
KEN1	91.4	6.8	23.9	1.341563	94.9	87.5	1.948761
KEN2	91.6	6.6	25.0	1.332019	95.2	85.9	1.951884

Table 12 Performance Using 16384 Byte 4-way Set Associative PRC

Trace	Average Read Access Time Speed Up over no PRC						
	256 byte	512 byte	1024 byte	2048 byte	4096 byte	8192 byte	16384 byte
EQNT	1.310371 4.9 %	1.308821 5.0 %	1.307791 5.04 %	1.306571 5.12 %	1.304870 5.25 %	1.301063 5.53 %	1.292856 6.12 %
KEN1	1.357973 4.60 %	1.356720 4.69 %	1.355939 4.75 %	1.354370 4.86 %	1.351975 5.03 %	1.348034 5.30 %	1.341563 5.76 %
KEN2	1.349544 4.71 %	1.348318 4.80 %	1.347330 4.87 %	1.345649 4.99 %	1.342989 5.18 %	1.338538 5.49 %	1.332019 5.95 %

Table 13 Summary of Read Performance Using 4-way Set Associative PRCs

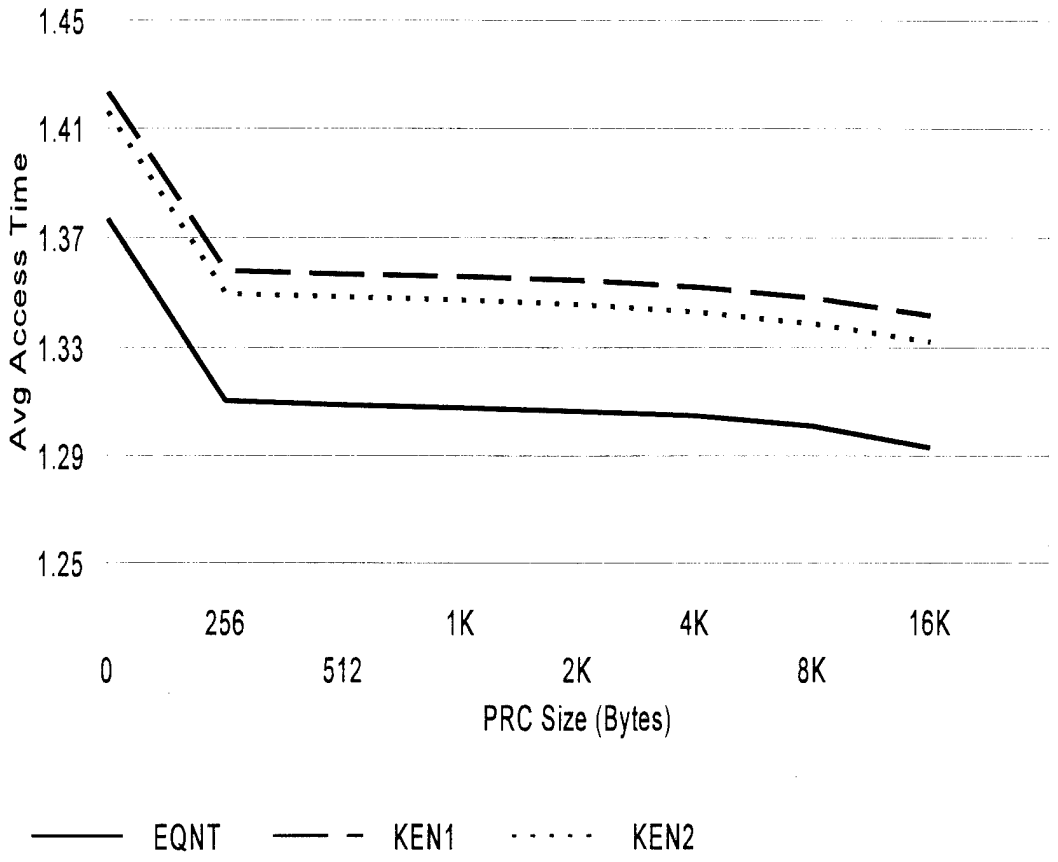

Figure 1 System Performance as a Function of PRC Size

The results show that, as would be expected, a larger PRC resulted in a higher PRC hit rate and a lower average read access time. Just as with the 256 byte PRC size, write performance was not significantly affected as PRC size increased. The higher PRC performance for the larger sizes can be attributed to the PRC retaining its predicted data longer and then acting like a second-level cache to provide the data when needed due to conflict or capacity misses in the first-level cache. However, as can be seen in Figure 1, the performance improvement as PRC size increased is small (4.7% average speedup with a 256 byte PRC increasing to 5.9% average speedup with a 16384 byte PRC). This means that a PRC 64 times larger only provided a 25% improvement in PRC performance. These results indicate that the benefit of the PRC comes mostly from its predictive nature and that the predicted data is normally used by the first-level cache soon after it has been fetched by the PRC. Therefore, the PRC acts to reduce the average memory access time chiefly by lowering the penalty for compulsory misses in the first-level cache. The small size of the PRC limits its ability to reduce the penalty for conflict or capacity misses in the first-level cache. Because of this, as long as the PRC has enough locations to hold the different prediction traces, its size has a minimal impact. Therefore, the 256 byte 4-way set associative PRC can achieve most of the performance gain of the 16384 byte PRC. These results also show that, although the different traces result in different average access times, the trends due to parameter changes are consistent among the traces. Therefore, for the rest of the simulations the longer KEN2 trace is not used due to the length of time the simulations would require.

3. Variation in PRC Set Associativity

The next set of simulations were to determine the effects of changing the PRC set associativity. Nominally, a higher degree of set associativity provides improved performance but at the expense of additional hardware [Ref. 1]. Fully associative PRCs were simulated for the smaller PRC sizes while direct mapped and 2 and 8-way associativities were simulated for all PRC sizes. Tables 14 through 20 detail the results of these simulations.

Trace	Average Read Access Time					Average Write Access Time				
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Fully Assoc	Direct Map	2-way S/A	4-way S/A	8-way S/A	Fully Assoc
EQNT	1.311851	1.310887	1.310371	1.310704	1.310262	1.93433	1.93418	1.93421	1.93435	1.93421
KEN1	1.358229	1.358137	1.357973	1.357758	1.357958	1.94875	1.94883	1.94879	1.948826	1.94878

Table 14 Effects of Changing PRC Set Associativity for 256 Byte PRC

Trace	Average Read Access Time					Average Write Access Time				
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Fully Assoc	Direct Map	2-way S/A	4-way S/A	8-way S/A	Fully Assoc
EQNT	1.310850	1.309347	1.308821	1.308914	1.308998	1.93431	1.93390	1.93444	1.93346	1.93430
KEN1	1.357581	1.356764	1.356720	1.356794	1.357635	1.94883	1.94873	1.94881	1.94882	1.94879

Table 15 Effects of Changing PRC Set Associativity for 512 Byte PRC

Trace	Average Read Access Time					Average Write Access Time				
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Fully Assoc	Direct Map	2-way S/A	4-way S/A	8-way S/A	Fully Assoc
EQNT	1.309216	1.308353	1.307791	1.308008	1.308248	1.93435	1.93415	1.93384	1.93395	1.934083
KEN1	1.36276	1.356064	1.355939	1.355768	1.356342	1.94883	1.94883	1.94878	1.94882	1.94876

Table 16 Effects of Changing PRC Set Associativity for 1024 Byte PRC

Trace	Average Read Access Time				Average Write Access Time			
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Direct Map	2-way S/A	4-way S/A	8-way S/A
EQNT	1.308026	1.306972	1.306571	1.306775	1.93462	1.93303	1.93458	1.93430
KEN1	1.354605	1.354319	1.354370	1.354343	1.94882	1.94882	1.94881	1.94812

Table 17 Effects of Changing PRC Set Associativity for 2048 Byte PRC

Trace	Average Read Access Time				Average Write Access Time			
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Direct Map	2-way S/A	4-way S/A	8-way S/A
EQNT	1.306335	1.305677	1.304870	1.304828	1.93445	1.93447	1.93414	1.933813
KEN1	1.352784	1.349630	1.351975	1.351759	1.94878	1.94883	1.94879	1.94882

Table 18 Effects of Changing PRC Set Associativity for 4096 Byte PRC

Trace	Average Read Access Time				Average Write Access Time			
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Direct Map	2-way S/A	4-way S/A	8-way S/A
EQNT	1.301886	1.303746	1.301063	1.300981	1.93432	1.93359	1.93411	1.934191
KEN1	1.349598	1.348663	1.348034	1.347499	1.94876	1.94882	1.94878	1.94879

Table 19 Effects of Changing PRC Set Associativity for 8192 Byte PRC

Trace	Average Read Access Time				Average Write Access Time			
	Direct Map	2-way S/A	4-way S/A	8-way S/A	Direct Map	2-way S/A	4-way S/A	8-way S/A
EQNT	1.302034	1.310189	1.292856	1.292966	1.93430	1.93464	1.93410	1.93413
KEN1	1.341690	1.341655	1.341563	1.337321	1.94874	1.94882	1.94876	1.94877

Table 20 Effects of Changing PRC Set Associativity for 16384 Byte PRC

As can be seen, increasing the set associativity normally resulted in a small decrease in average access times, but the effect varied depending on the size of the PRC, the associativity, and the particular trace being used. However, on the average, there was only a change of 0.13 % in the average read access time due to associativity changes. This can be attributed to the fact that data read into the PRC is either used quickly by the first-level cache or is not used at all. Therefore, any increase in set associativity has minimal impact since conflict misses in the PRC are not common. As a result, the PRC should use the easiest set associativity to implement (normally direct mapping) to minimize its hardware complexity and size.

4. Variation in PRC Miss Allocation Policy

The PRC can operate with any of the three common cache miss allocation policies (LRU, Random, FIFO). This set of simulations were run to determine the effects on average access time for changes in the PRC miss allocation policy. These simulations were done using PRC sizes of 256, 4096, and 8192 bytes and the results are shown in Tables 21 through 23. As can be seen, the change from LRU resulted in a very small access time degradation. This again is attributed to the fact that data in the PRC is either

used soon after it is read in or not at all. Therefore, the simplest replacement algorithm would be recommended to reduce system complexity.

Trace	Read Average Access Time			Write Average Access Time		
	LRU	Random	FIFO	LRU	Random	FIFO
EQNT	1.310262	1.312604	1.315710	1.934211	1.934441	1.934121
KEN1	1.357958	1.358787	1.362811	1.948782	1.948831	1.948812

Table 21 Effects of PRC Miss Allocation Policy with 256 Byte Fully Associative PRC

Trace	Read Average Access Time			Write Average Access Time		
	LRU	Random	FIFO	LRU	Random	FIFO
EQNT	1.304870	1.305913	1.306522	1.934135	1.934533	1.934568
KEN1	1.351975	1.352489	1.354097	1.948786	1.948806	1.948803

Table 22 Effects of PRC Miss Allocation Policy with 4096 Byte 4-way Set Assoc. PRC

Trace	Read Average Access Time			Write Average Access Time		
	LRU	Random	FIFO	LRU	Random	FIFO
EQNT	1.301063	1.302155	1.302231	1.934111	1.934378	1.933754
KEN1	1.348034	1.349315	1.350633	1.948784	1.948806	1.948806

Table 23 Effects of PRC Miss Allocation Policy with 8192 Byte 4-way Set Assoc. PRC

5. Variation in the Maximum PRC Read in Buffer to Continue

The Max PRC Read in Buffer parameter determines how often an in-progress PRC read gets canceled when new cache read requests are added to the read buffer. As explained in Section II.B.3, the setting of this parameter is a balance between getting cache read misses started as early as possible without wasting too much memory bandwidth on incomplete PRC read requests. This set of simulations was done using a 256 byte fully associative PRC and a 1024 4-way set associative PRC and compared the

performance when the Max PRC Read in Buffer was set to 4 (the value used in all other simulations) and when it was set to 8. As can be seen from Table 24, increasing the size of this parameter resulted in delaying cache read requests and slightly poorer system performance.

Trace	256 byte PRC		1024 byte PRC	
	Max Size = 4	Max Size = 8	Max Size = 4	Max Size = 8
EQNT	1.310262	1.310318	1.307791	1.308179
KEN1	1.357958	1.359335	1.355939	1.356460

Table 24 Effects of Varying Max PRC Size to Continue

6. Variation in PRC Write Policy

All of the simulations so far have used a PRC write policy of write through where the PRC is updated at the same time as main memory (if the data was already in the PRC). To determine the effects of a write around policy on system performance, a set of simulations was run using PRC sizes of 256 and 1024 bytes. The results of these simulations are given in Table 25. They show that the difference between the write around policies had different effects depending on the trace used and the PRC size. The small and variable nature of the change seems to indicate that there is no clear advantage to either method.

Trace	256 byte PRC		1024 byte PRC	
	Write Through	Write Around	Write Through	Write Around
EQNT	1.310262	1.307876	1.307791	1.307690
KEN1	1.357958	1.358217	1.355939	1.355946

Table 25 Effects of Varying PRC Write Policy

7. Effects of Other PRC Parameters

There are two other PRC parameters that could affect PRC performance. They are *UsePRCOnWriteMiss* and *DropPRCOnSecondMiss*. Based on the results of preliminary testing, all of the simulations so far have had *UsePRCOnWriteMiss* = No and *DropPRCOnSecondMiss* = Yes. Tables 26 and 27 show the effects of changing these values on system performance. *UsePRCOnWriteMiss* being set to Yes has a significant effect on performance for the 256 byte fully associative PRC, probably due to the fact that the writes to memory are not consistent with the data being read. This means that starting a PRC trace based on a write does not provide data that will be used by a future cache read and may even write over a valid prediction trace. This effect is much less pronounced in the 1024 byte 4-way set associative PRC. Changing *DropPRCOnSecondMiss* has negligible effect on system performance. This is due to the large eight request read buffer and the mechanism for handling request priorities. Even if a PRC request is slipped twice, it does not impact future reads because it will stay at the bottom of the read buffer until all other read requests are complete.

Trace	256 byte PRC		1024 byte PRC	
	Use = No	Use = Yes	Use = No	Use = Yes
EQNT	1.310262	1.350188	1.307791	1.307807
KEN1	1.357958	1.373540	1.355939	1.370687

Table 26 Effects of Varying *UsePRCOnWriteMiss*

Trace	256 byte PRC		1024 byte PRC	
	Drop = Yes	Drop = No	Drop = Yes	Drop = No
EQNT	1.310262	1.309977	1.307791	1.307808
KEN1	1.357958	1.358575	1.355939	1.358557

Table 27 Effects of Varying *DropPRCOnSecondMiss*

D. RESULTS MODELING A SECOND-LEVEL CACHE

To allow comparison of system performance with a PRC to that of a system with a second-level cache, simulations of a 16384 byte 4-way set associative second-level cache were done using SACS21. The results of these simulations are shown in Table 28 and show that a second-level cache of this size can provide a significant performance increase. The cache size was chosen to be comparable with the PRC sizes tested earlier. The second-level cache in these simulations had an access time of one clock cycle. This speed would only be possible on a small on-chip cache. The results of using a more realistic access time of two clock cycles is shown in Table 29. With this assumption, the performance of an 8192 PRC is nearly comparable with a cache twice its size. A simulation of an 8192 byte 4-way set associative second-level cache with a one cycle access time was done even though this size cache would not normally be used with an 8192 byte first-level cache. The results of this simulation are shown in Table 30. They indicate that this size of second-level cache provides less of a speedup than a 256 byte PRC. This confirms that the predictive nature of the PRC is what provides the greatest benefit in reducing average memory access time.

The increase in average write access time over only a first-level cache is attributed to the reduced number of buffer hits on writes since all writes now go through the second-level cache. Therefore, any writes that miss the first level cache must wait the second-level access time before possibly getting a hit in the second-level cache. Without a second-level cache, these writes could have hit the buffer as soon as they were being written out by the first-level cache.

	Average Read Access Time	Speed Up	Average Write Access Time	Speed Up
EQNT	1.279652	7.08 %	1.936717	slower
KEN1	1.319244	7.32 %	1.960368	slower
KEN2	1.314057	7.22 %	1.962551	slower

Table 28 Performance Using a 16 Kbyte Second-Level Cache with 1 Cycle Access

	Average Read Access Time	Speed Up	Average Write Access Time	Speed Up
EQNT	1.300383	5.56 %	1.936729	slower
KEN1	1.343165	5.64 %	1.960369	slower
KEN2	1.339284	5.44 %	1.962555	slower

Table 29 Performance Using a 16 Kbyte Second-Level Cache with 2 Cycle Access

	Average Read Access Time	Speed Up	Average Write Access Time	Speed Up
EQNT	1.334967	3.06 %	1.933316	0.04 %
KEN1	1.377860	3.21 %	1.956252	slower

Table 30 Performance Using a 8 Kbyte Second-Level Cache with 1 Cycle Access

IV. CONCLUSION

A. SUMMARY OF RESULTS

The simulations run in this thesis proved that a PRC can provide a significant performance improvement to a memory hierarchy containing only a first-level cache. Even a small (256 byte) PRC provided a 4.7 % speedup while the 16384 byte PRC provided a 5.9 % speedup. A 16384 byte second-level cache simulation resulted in a 7.2 % speedup that is better than any of the PRC results. However, the cache simulation assumed that the second-level cache could provide data on a cache hit in only one processor clock cycle. This assumption may not be true for this size of a cache, and would definitely not be true for an external second-level cache. Using a second-level cache hit access time of two processor clock cycles resulted in a 5.6 % speedup; approximately equal to the speedup of an 8192 byte PRC (5.5 % speedup). A small PRC gets its performance improvement with very little hardware such that it could easily be included on the microprocessor chip, and provide hit data in only one clock cycle. Therefore, in an actual implementation, a 256 byte PRC might outperform a much larger second-level cache. The simulation results also that the PRC associativity is not a major effect as long as there are enough different sets to track all prediction traces without overwriting other valid prediction traces. A PRC configured in this manner should then be fetching data from main memory just before it being needed by the first-level cache. This allows the PRC to reduce the miss penalty for first-level cache compulsory misses and thereby reduce the average memory access time of the system. As can be seen from Figure 2, the 256 byte direct mapped PRC met this goal and provided the best price to performance ratio since its performance was almost identical to the 8192 byte PRC while using much less hardware.

The simulation results also showed that the PRC block replacement policy (LRU, random, or FIFO) had minimal impact on PRC performance. This result allows the

system designer to choose the easiest method to implement when incorporating a PRC into a system design. The decision on whether to use the PRC on cache write allocates is a harder one as the results were less clear, however, with a small PRC these reads pollute the prediction traces and lower performance. In this case, the PRC should only be used for cache read misses. Variations in PRC write policy had little effect and the simplest method to implement the PRC should be chosen.

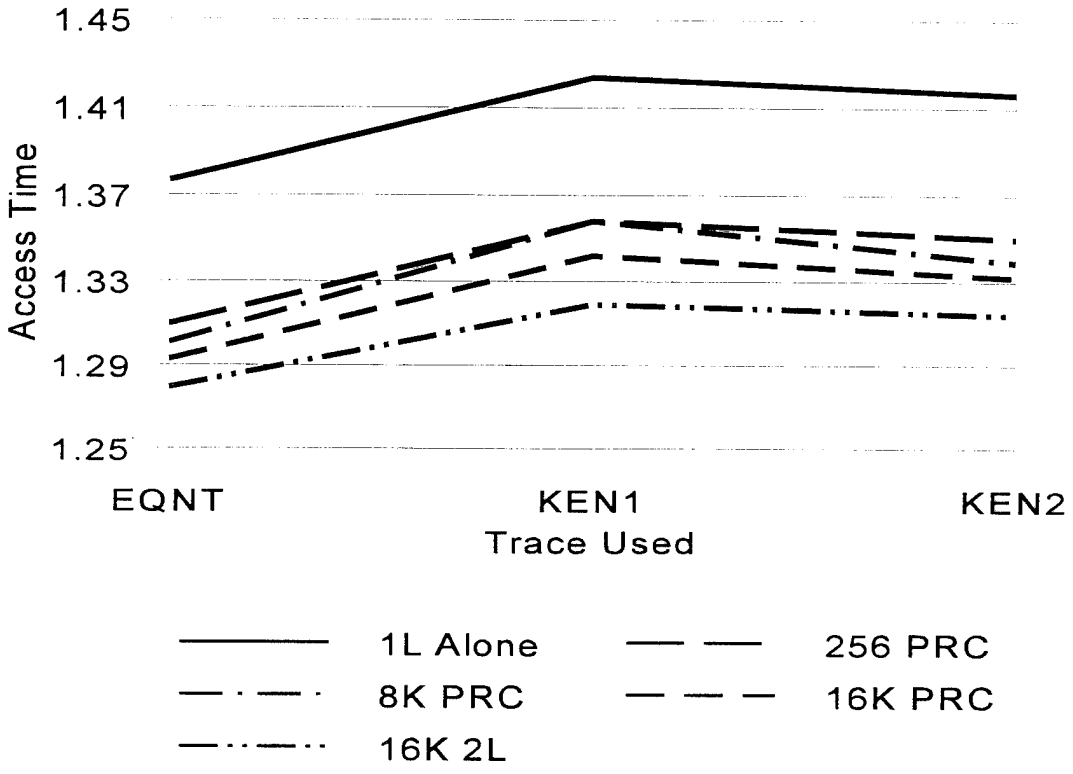

Figure 2 Comparison of Average Read Access Times For Different Configurations

B. RECOMMENDATIONS

A PRC provides an easy way for a microprocessor system designer to gain a performance improvement without a major hardware investment. It is also a method to reduce the miss penalty associated with compulsory cache misses (the initial filling of the

first-level cache) since a PRC will retrieve some requests before the cache requires them. Since a small PRC provides almost the same performance improvement as a large one with much lower cost, the optimum configuration to use would be a 256 byte fully associative PRC coupled with an eight request read buffer. The PRC should be designed to be bypassed on cache write allocate reads and should use the simplest block allocation policy available. This arrangement would gain the maximum performance increase for the lowest overall cost.

Although a large second-level cache outperforms a PRC, it does not mean that there is no use for a PRC in microprocessor design. Many designs cannot use a large second-level cache, whether because of chip area constraints, power constraints, or system cost constraints. These systems may be embedded microprocessors, portable computers or space applications. In these cases, a PRC would give enhanced performance with very little additional hardware or power consumption.

Another possible use of a PRC is to place it between the first-level and second-level caches where its role would be to reduce the compulsory miss penalty. In this application, a very small PRC could be used to bring predicted data into the second-level cache where it would be available when a first-level cache compulsory miss occurs. The PRC itself would only need to store the information necessary to compute the predicted addresses. This type of PRC might even be software controlled so that it only functioned when the first-level cache is being initially filled with data following a task switch and is then turned off once both caches have been filled with new data. This type of implementation could be simulated using a modification to the SACS2 program to incorporate both a PRC and a second-level cache. The resulting system would combine the best of the PRC and second-level cache implementations and could be used on very high-end microprocessors that need every possible method to speed up memory accesses.

Further investigation into the use of a PRC to enhance memory system performance needs to be done. Use of a PRC in lieu of a second-level cache shows great

promise to provide higher performance for less power and cost while incorporating a PRC in a multi-level cache hierarchy may result in a way to reduce the impact of compulsory misses. Using a modified version of the SACS2 program, additional simulations could be run to verify that the PRC most affects the compulsory miss penalty and has little effect on the conflict or capacity miss penalties. Also, testing with an actual PRC in conjunction with a microprocessor containing a first-level cache could be performed to obtain actual performance measurements to validate simulation results.

LIST OF REFERENCES

1. Patterson, D. A. and Hennessy, J. L., *Computer Organization and Design: The Hardware/ Software Interface*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1994.

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

3. Crawford, J. H., "The i486 CPU: Executing Instructions in One Clock Cycle," *IEEE Micro*, pp 27-36, February 1990.

4. Colwell, R. P. and Steck, R. L., "A 0.6um BiCMOS Processor with Dynamic Execution," *ISSCC Proceedings*, pp 176-177, February 1995.

5. Edmondson, J. H. and others, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro*, pp 33-43, April 1995.

6. Fouts, D. J. and Billingsly, A. B., "Predictive Read Caches: An Alternative to On-Chip Second-Level Cache Memories," *Journal of Microelectronic Systems Integration*, pp 109-121, June 1994.

7. Smith, W. G., "SACS: A Cache Simulator Incorporating Timing Analysis with Buffer and Memory Management," Master's Thesis, Naval Postgraduate School, Monterey, CA, 1994.

8. Grimsrud, K. and others, "BACH: A Hardware Monitor for Tracing Microprocessor-Based Systems," unpublished paper, Brigham Young University, Provo, UT, February 1993.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
4.	Professor D. J. Fouts, Code EC/Fs Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
5.	LCDR Robert W. Miller, USN 3808 Outrigger Drive Edgewater, Maryland 21037	1