**Combining Selective Cache Line Replacement and Active Management for Data Caching**

By

YAN TING BETTY CHEN

B.S. (University of California at Davis) 2003

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_________________________________

Professor Matthew Farrens (Chair)

_________________________________

Professor Charles U. Martel

_________________________________

Professor Venkatesh Akella

Committee in Charge

2005

# Combining Selective Cache Line Replacement and Active Management for Data Caching

Copyright 2005

By

Yan Ting Betty Chen

**Acknowledgements**

I would like to express my gratitude to my advisor, Professor Matt Farrens, for his guidance, encouragement, and understanding. Not only did he provide me helpful advice on my research, but also served as my mentor. He is friendly and easy-going. I feel comfortable talking with him as a student and as a colleague. Credits also go to my thesis committee members, Professor Chip Martel and Professor Venkatesh Akella, for their insightful comments on this thesis. I would like to thank all people in the Architecture Lab for their helpful technical assistances and thoughtfulness.

Special thanks goes to all my friends for their fruitful discussion and valuable support. I treasure the wonderful time we spent together and the memory we shared for the past years. Last, I appreciate everyone in my family for their unconditional love and caring. They have given me inspiration and influenced my education throughout these years.

## Abstract

Due to the increase in computer processing power over the past decades, computer performance has improved dramatically. However, relatively slower growth in memory speed has led to memory accesses becoming one of the bottlenecks in high performance machines. Reducing the number of misses that occur in on-chip caches can improve the machine performance significantly. It is also important to minimize memory bandwidth. The selective line replacement algorithm proposed in [24][26] can reduce memory bandwidth with minimal impact on cache miss rate. Recent studies show multi-lateral caches using active data management schemes which include reuse information [17][22][25] can effectively reduce the miss ratio better than larger single structure caches in many cases.

This thesis studies the three types of hybrid caches that incorporate the Non-Temporal Streaming (NTS) and Program Counter Selective (PCS) multi-lateral caches with the selective line replacement algorithm. The Intel Atom + mlcache tool sets are used to evaluate the performance of these caches on a subset of the SPEC2000 benchmarks. The three hybrid caching schemes, including NTSCNA, NTSPCS, and NPCNA, show improvement in performance for various benchmarks compared to a single structure cache. Among the three proposed schemes, the NTSPCS performs the best, followed by the NTSCNA and NTSPCS schemes. In fact, the NTSPCS cache can even outperform both the NTS and the PCS caches. Although the NTSCNA scheme does not outperform the NTS and the NTSPCS cache, it can perform almost as well as a single structure cache that is twice its size and it can save a significant amount of memory bandwidth.

# Contents


**1 Introduction and Background ...... 1**
1.1 Introduction ...... 2
1.2 Background ...... 4
1.3 Miss Rate ...... 5
1.4 Miss Penalty ...... 6

**2 Cache Behaviors ...... 9**
2.1 Simulation Tools ...... 10
2.2 Benchmarks ...... 11
2.3 Data References and Miss References Characteristics ...... 14
2.4 Selective Cache Line Replacement ...... 18
2.4.1 Static Method ...... 19
2.4.2 Dynamic Method ...... 20

**3 Multi-lateral Caches ...... 24**
3.1 NTS Data Cache ...... 26
3.2 PCS Data Cache ...... 29
3.3 Proposed Models - Hybrid Cache Models ...... 31
3.3.1 NTSCNA Data Cache ...... 32
3.3.2 NTSPCS Data Cache ...... 34
3.3.3 NPCNA Data Cache ...... 37
3.4 Simulation Environment ...... 38

**4 Results ...... 41**
4.1 Miss Ratio ...... 41
4.2 Cache Speedup ...... 45
4.3 Memory Bandwidth ...... 47
4.4 Counter Configuration Impacts ...... 49
4.4.1 NTSCNA Changes ...... 50
4.4.2 NTSPCS Changes ...... 51
4.4.3 NPCNA and NPCNA-Prev Changes ...... 52

**5 Conclusions and Future Work ...... 54**

**Bibliography ...... 57**

**Appendix A ...... 60**

# Chapter 1

# Introduction and Background

Due to the increase in computer processing power over the past decades, computer performance has improved dramatically. However, the relatively slower growth in memory speed has caused memory accesses to become a major bottleneck in high performance machines. Accessing off-chip memory is extremely slow, and has led to the extensive use of a hierarchical memory structure in modern processors. Almost all processors today have at least the first level of the memory hierarchy on-chip, and some have even more than that. A miss in an on-chip cache has a very long latency measured in terms of processor cycles, so clearly as the number of cache misses increases, the resulting miss penalty will substantially lower processor utilization. In addition, the problem becomes more severe for multiple-issue processors since the number of cache misses can increase as the issue width of the processor increases. As technology advances, processors may be able to issue more and more instructions per cycle, and the miss penalty will become more and more of a problem. Therefore, finding ways to reduce the access time between memory and the processor is crucial.

## 1.1 Introduction

Over the years, different techniques to reduce the gap between CPU and memory speeds have been introduced. Multilevel caches [4][15][17][22][25][26] are examples of one of the more intuitive ways to improve cache performance. Since a level one cache is added to machines to help reduce the memory access time, adding yet another level of cache can logically help further reduce the miss penalty. Adding a large second level cache can eliminate many accesses to the main memory because this additional cache will be able to capture most of the frequently used data.

Another strategy is to prefetch data into the cache before it is referenced by the CPU. Data prefetching can greatly improve the memory hit rates, and thus can benefit the overall performance of the processor. Prefetching can be done by the hardware or by the software. Between the two prefetching techniques, hardware prefetching is the most popular approach [2]. The main reason for its popularity is the fact that improvements in technology greatly increase the feasibility of a proposed hardware implementation. The hardware approach also has access to program run-time information (such as the hits and misses of memory references) available at the hardware level [7]. However, the software approach, explained by Mowry et al. in [13], is still a popular scheme (due to its low cost). The high level description of this scheme is to insert prefetch instructions into codes that execute dense matrix codes. For software prefetching, added instructions can affect compiler optimization performance as described in [21][29].

Lock-up free (non-blocking) caches are a type of pipelined cache design that do not require the processor to stall on cache misses until the misses are resolved [5]. They do not reduce the number of misses a cache will experience, but they allow the data cache

to continue servicing processor requests during cache misses. This reduces the cache miss penalty by lowering the stalls on cache misses.

Studies have also shown that multi-lateral caches with intelligent cache management schemes can reduce cache miss ratios and average access times. These multi-lateral cache designs allow two or more disjoint same level caches that contain different data sets to be checked in parallel on each memory access. According to Tam et al. [23][25], multi-lateral caches such as Victim cache, Assist Cache, NTS caches, and PCS caches have equal or even better performance than larger caches that have single cache designs [11][15][16][17].

The Victim cache is the simplest of these, and can intelligently manage the cache without reuse information. It uses a very small fully associative cache (or buffer) located between a cache and its refill path to store recently evicted blocks. These blocks can be quickly retrieved if needed again without having to access the main memory. The Assist cache, on the other hand, employs a small cache to store data blocks before they enter the larger (main) cache. An evicted block from the smaller cache will be stored in the main cache. Unlike these first two cache schemes, NTS caches make use of the effective address (EA) of the block to deal with data replacement during a miss, whereas PCS caches use the program counter (PC) of the instruction that generated the request for the block. The details of the NTS and PCS caches will be discuss in a later chapter.

Multi-lateral cache designs have a unique way of reducing cache misses, and this thesis will try to explore more fully the potential of this type of cache design. The main design focus, however, will be on designs that rely on utilizing previous use information to manage data caches. Since the two strategies that use the EA or the PC both improve

performance effectively, the potentials of caches that use both the EA and PC together will be studied. Additionally, because a small number of instructions account for a large number of references (and misses), incorporating the selective cache line replacement algorithm (introduced in [24][26]) into the multi-lateral cache designs will be examined.

## 1.2 Background

Reducing the gap between the CPU speed and main memory access time is critical for increasing the machine performance. Caches are used extensively to reduce this gap, and one of the key factors to maximize cache performance is to minimize the miss rate (or maximize the hit rate). Although the miss rate is not the ideal metric for evaluating memory hierarchy performance (as pointed out in [10]), it has a close relationship with another better metric, the average memory access time. Different from miss rate, which is not dependent on hardware speed, average memory access time can be measured in either absolute time units or the number of CPU clock cycles it takes to read from and write to memory. The average memory access time calculation includes the hit time, miss rate, and miss penalty information. As stated in [10], its formula is the following:

$$\textit{Average Memory Access Time} = \textit{Hit time} + \textit{Miss rate} * \textit{Miss penalty}. \quad (1)$$

Equation (1) shows that the average memory access time of level one cache is directly proportional to the miss rate, and lowering the miss rate while keeping all other terms the same will lead to better cache performance. The same relationship applies to the miss penalty, where a lower miss penalty implies performance enhancement.

The overall performance includes more than just the average memory access time. Minimizing the CPU execution time is also very important, and the memory stall cycle metric is an important element in this calculation. Again, miss rates and miss penalty affect the memory stall cycle calculation directly, as can be seen in equation (2).

$$\begin{aligned} \text{Memory stall cycles} &= \text{Number of misses} * \text{Miss penalty} \\ &= \text{Reads} * \text{Read miss rate} * \text{Read miss penalty} + \\ &\quad \text{Writes} * \text{Write miss rate} * \text{Write miss penalty.} \end{aligned} \tag{2}$$

These two equations indicate that we need to concentrate on reducing both the number of misses and the miss penalty in order to optimize the memory hierarchy performance. We will explore these two elements in greater detail next.

## 1.3 Miss Rate

In general, cache misses can be classified into 3 types - capacity, conflict, and compulsory misses. Capacity misses refer to misses caused by having more lines of frequently accessed data than the number of lines that the cache can hold. Due to the size limit of the cache, active data will be evicted from the cache once the cache is full of more recently used data. This will cause a miss the next time the processor issues a request for the data that was evicted. Conflict misses have to do with block placement strategies and are more likely to occur with low associativity caches. Conflict misses occur when too many active addresses are mapped to the same location (cache line) in the cache. Because of this mapping, some useful blocks might be discarded, leading to cache misses when these evicted blocks are requested later on. Finally, compulsory misses occur independent of the cache associativity or size. During the processor's first access to the cache, for example, the data cannot yet be in the cache and will be brought into the

cache after the first request. Compulsory misses are also known as cold-start or first-reference misses.

Among the three types of misses, Hennessy and Patterson in [10] show that the conflict miss rate is inversely proportional to cache associativity such that increasing the associativity will lower the number of conflict misses. However, capacity misses can become a major problem for fully-associative caches. Finally, [10] points out that compulsory miss rates are very low for long-running programs such as SPEC2000.

Most memory system designers concentrate on reducing capacity and conflict misses, since compulsory misses can only be reduced using previously described techniques like prefetching. Enlarging the caches is the only real solution for reducing capacity misses, but a larger cache will mean longer hit times and higher hardware costs. Fully associative caches can eliminate conflict misses completely, but these caches require a lot of extra hardware and therefore are costly and have longer hit times as well (which can degrade the overall processor performance). Furthermore, various studies explore [14][26][27] the relationships between increasing cache sizes, cache associativities, and cache access times. They conclude that cache access times increase significantly once the cache size exceeds 16K bytes. Changing from a direct-map (1-way associative) cache to a 2-way associative cache can increase access time also.

## 1.4 Miss Penalty

Decreasing the miss rate is crucial in order to enhance processor performance, but minimizing miss penalty is equally important. The miss penalty refers to the time it takes to move a block between one cache level and the next. As the gap between the processor

and memory speeds continues to increase, the values of the miss penalty climb quickly as well. Many of the methods used to lower the miss penalty have focused on using prefetching techniques. As mentioned in the previous chapter, prefetching techniques include hardware and software, with the hardware approach being more popular. One of the most basic hardware prefetching mechanisms is one-block-lookahead (OBL) or always prefetch as described in [19]. For OBL, when an application makes a reference to a block located at i, the block at i + 1 will be requested as well.

The stream buffer proposed by Jouppi [12] is an improved version of the OBL technique. Instead of prefetching only one block as in OBL, the stream buffer prefetches one or more blocks. The stream buffer is implemented as a FIFO queue that holds successive cache miss blocks. These stream buffers are located between the lowest level cache and the next higher level cache. Having a cache miss but a hit in the stream buffer is much more efficient than sending the request to the next level cache, which can take many CPU cycles to access. This method shows that there is a high probability of future reference to the next block, because the stream buffer is able to capture many misses as pointed out in [10].

Another kind of hardware prefetching takes into account recent reference history when issuing a prefetch. The Stride Prediction Table (SPT) technique described by Fu and Patel in [8] improved the performance of vectorized numerical programs by reducing vector cache misses. According to [7], such a scheme exploits block reference probabilities and will work well with programs that have regular memory access patterns.

Although these prefetching techniques can reduce the miss penalty, these latency-tolerance techniques may be ineffective because of memory bandwidth constraints.

Prefetching can create contention in the memory system by fetching more data than is actually needed [5]. In addition, Ding et al. in [8] studied programs like the Linpack kernel (dmxpy) and found that the data from memory cannot be delivered fast enough to keep the CPU busy due to insufficient memory bandwidth. Consequently, memory performance measures not only depend on miss rates but also on memory bandwidth [8].

A study by Abraham et al. [1] found that a small number of instructions produce a disproportionately large number of data references (and misses). Their idea will be described in greater detail in the next chapter. This thesis will be presented in the following order: in Chapter 2, I will verify the fact that a small number of instructions cause large number of data references and misses on a subset of SPEC2000 benchmarks. In addition, the same chapter will describe the selective cache line replacement algorithm. Chapter 3 will explain multi-lateral NTS and PCS caches, and my newly proposed cache schemes. I will discuss in detail the results of the experiments ran on different benchmarks in Chapter 4, and present my conclusion in Chapter 5.

# Chapter 2

# Cache Behaviors

In order to increase our understanding of cache and program behavior, Tyson, et al. [26] and Abraham, et al. [1] examined the primary causes for most misses in a program. Using cache simulators, they studied the relationship between load/store instructions and data references as well as the behavior of load/store instructions with respect to misses on load/store machines. The simulations not only measured the overall miss rates of the benchmarks, but also the association of an individual instruction with the number of data accesses and the number of misses that instruction incurred. The results presented in both papers show that a very small number of instructions are responsible for a large percentage of data references and cache misses. Abraham et al. proposed that selectively scheduled prefetches can reduce overall cache miss ratios. This technique prefetches data referenced by instructions that cause a high number of misses.

Knowing that this scheduling strategy reduced the miss ratio but would be limited by the available memory bandwidth for data prefetching, Tyson, et al. decided to explore modifying this method for use in data cache management instead of data prefetching. They felt that caches could be managed more efficiently and miss rates could be lowered using a variant of the selective line scheduling idea. The results of cache simulations

with their modified selective line cache line replacement algorithm demonstrated reassuring cache performance. However, the SPEC89 and SPEC92 benchmarks used in [1] and [26] respectively are very old and are not memory-intensive (they do not exercise the memory system much). To ensure that a small number of instructions account for a large number of data references and misses in more modern programs, I have ran their simulations again with a newer set of benchmarks (the SPEC2000 suite).

## 2.1 Simulation Tools

To show the correlation between the different individual instructions and data caching behavior, I employed the Intel Analysis Tools for Object Modification (Intel ATOM) program available at [11] to gather the dynamic data [20]. Intel ATOM is a flexible program analysis system that allows users to create various tools to more completely evaluate program behaviors. For data cache simulations, Intel ATOM includes a 32 byte/line 16K direct-mapped cache simulation tool. During each simulation run, each of the PC's that generate a data reference will be collected, along with their data reference frequency and miss frequency. Each time a data reference or cache miss occurs, the reference or miss frequency corresponding to this PC will be increased by one. By modifying the provided cache simulator, these values can be determined throughout each run. After processing all of the collected data, statistics on cache behavior can be compiled in a straightforward manner.

## 2.2 Benchmarks

The cache simulations were run on fourteen different SPEC2000 benchmark programs, including eight integer and nine floating-point programs [18]. All benchmarks were built using SPEC2000's default peak performance compilation settings (various compiler optimization flags are set for individual benchmarks aiming to bring forth their best performance). In order to reduce the execution time of the simulations, the benchmarks were run using test inputs, but all are run to completion. The following subset of SPEC2000 benchmarks will be used through out all simulations.

**SPEC2000 Integer Benchmark**

**164.gzip** The SPEC version of this data compression benchmark performs its compression and decompression only in memory in order to separate CPU and memory subsystem usage. Its compression algorithm is based on the Lempel-Ziv code.

**175. vpr** 175. vpr (Versatile Place and Route) serves as a type of integrated circuit computer-aided design program and this VPR is designed for placement and routing in Field-Programmable Gate Arrays (FPGA). This program takes in a netlist file that describes the initial circuit placement and routes, and also a file that describes the FPGA architecture. Based on these input files, the program first generates how the circuit will be placed into the FPGA. The final circuit routing will be generated by a second run with input files net.in, arch.in, and place, which provide assignments of the positions to each circuit element in the netlist.

**176.gcc** 176.gcc is based on gcc Version 2.7.2.2, and is an optimizing compiler that produces Motorola 88100 assembly code. The benchmark's inlining heuristic was

modified to generate more inline code so that more analysis can be done on source code inputs.

**181.mcf** This benchmark calculates a solution to the single-depot vehicle scheduling problem for public transportation companies. The solution will try to minimize the number of vehicles used based on all timetable trips. This minimum-cost flow network problem uses a specialized version of the network simplex algorithm.

**186.crafty** Crafty simulates a Chess game and uses a 64-bit word, but can be run on 32 bit machines by altering the data type. It takes in a chessboard layout and outputs a set of best possible moves based on searching in each level of the game tree.

**197.parser** Parser serves as an English syntactic parser, which uses labeled links to determine the syntactic structure of the input sentence. The program uses a 60000-word dictionary that contains syntactic construction information.

**255.vortex** VORTEx, an acronym for Virtual Object Runtime Expository, is a single user object oriented database program. This program basically records transactions such as create, delete, and search of entries that occur within the database during each run.

**300.twolf** This benchmark, derived from the TimberWolfSC program, provides solutions for placement and global connections of groups of transistors in a microchip. The program uses the concept of placing groups of transistors in rows to share connections for placement design. Using this design, the program can use a constructive algorithm to interconnect the transistors on a microchip. This program is the most numeric intensive benchmark in the SPEC suit. This benchmark’s emphasis

is on inner loop calculations that cause cache misses while traversing through memory.

**SPEC2000 Floating Point Benchmarks**

**168.wupwise** This Wuppertal Wilson Fermion Solver is used in part of the lattice gauge theory simulation in quantum chromodynamics. In the simulation, the calculations of the quark propagators are required and this takes an enormous amount of computing power. The Wupwise can compute quark propagators by solving the inhomogeneous lattice-Dirac equation.

**171.swim** Swim simulates shallow water modeling in the field of meteorology and does intensive floating point computing. This program uses the model from "The Dynamics of Finite-Difference Models of the Shallow-Water Equations", by Robert Sadourny, J. ATM. SCIENCES, VOL 32, NO 4, APRIL 1975.

**172.mgrid** Mgrid calculates three dimensional potential fields and is a multigrid solver.

**173.applu** Applu is a program for Computational Fluid Dynamics and Computational Physics, which computes parabolic/elliptic partial differential equations.

**177.mesa** Mesa is a library that supports three-dimensional graphics. Given data that describes an object in two dimensions, a three dimensional object will be created with scalar data mapped to height and a two-dimensional image file will be produced.

**179.art** This program implements the Adaptive Resonance Theory 2 neural network and is used to recognize objects in thermal images. First the program will be trained with a specific object, and will try to match the object in the image with the specific object after the program finishes learning.

**183.equake** With input data of unstructured topology, seismic event characteristics, and structure of sparse system matrix, this program will output the pattern of ground motion due to a specific seismic event occurring in a large basin by simulating the propagation of elastic waves.

**188.ammp** Ammp simulates the molecule dynamics of large systems of protein-inhibitor complex molecules embedded in water. Given the initial velocity and coordinates of the atoms, the program will generate the final energy level of the atoms.

**301.apsi**. This benchmark predicts weather by computing problems related to temperature, wind, velocity and distribution of pollutants.

## 2.3 Data References and Miss References Characteristics

By profiling these benchmarks, a similar pattern regarding the number of instructions causing data references and misses can be observed. Such a pattern shows that many different types of benchmarks exhibit similar characteristics. This finding also indicates that the selective line replacement caching strategy proposed in [24][26] will work for a wide range of programs. Table 2.1 shows the relationship between the number of instructions and the data reference patterns of the instructions. The information collected for each benchmark will be displayed in their rows, subdivided by columns. The first column of the table contains the names of the benchmarks and the following columns show the total number of references and number of unique dynamic instructions in each benchmark. The rest of the columns in the table give more detail about instruction reference patterns. These columns are divided by the total percentage of data references, including 75%, 90%, 95%, and 99%. In each percentage category, the

| SPEC2000 | **Number of references** | **# of inst** | **Percent of References** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 75% | | 90% | | 95% | | 99% | |
| | | | # of inst | % of total | # of inst | % of total | # of inst | % of total | # of inst | % of total |
| **ammp** | 2297068334 | 3783 | 336 | 8.88% | 496 | 13.11% | 575 | 15.20% | 1273 | 33.65% |
| **applu** | 197013988 | 6741 | 1435 | 21.29% | 2026 | 30.05% | 2366 | 35.10% | 3922 | 58.18% |
| **apsi** | 3131987543 | 6004 | 519 | 8.64% | 729 | 12.14% | 970 | 16.16% | 1983 | 33.03% |
| **art** | 1304143111 | 745 | 37 | 4.97% | 73 | 9.80% | 98 | 13.15% | 131 | 17.58% |
| **crafty** | 1867048548 | 10952 | 1061 | 9.69% | 1855 | 16.94% | 2439 | 22.27% | 3757 | 34.30% |
| **equake** | 298995789 | 1343 | 183 | 13.63% | 293 | 21.82% | 359 | 26.73% | 523 | 38.94% |
| **gcc** | 434788184 | 45345 | 2978 | 6.57% | 6599 | 14.55% | 9838 | 21.70% | 17731 | 39.10% |
| **gzip** | 25642815878 | 1298 | 73 | 5.62% | 101 | 7.78% | 120 | 9.24% | 272 | 20.96% |
| **mcf** | 10363355 | 416 | 11 | 2.64% | 29 | 6.97% | 46 | 11.06% | 92 | 22.12% |
| **mesa** | 5188453 | 647 | 10 | 1.55% | 22 | **3.40%** | 35 | **5.41%** | 62 | **9.58%** |
| **mgrid** | 13693863904 | 920 | 67 | 7.28% | 89 | 9.67% | 140 | 15.22% | 201 | 21.85% |
| **parser** | 967190620 | 6190 | 446 | 7.21% | 1014 | 16.38% | 1464 | 23.65% | 2580 | 41.68% |
| **swim** | 120086513 | 246 | 35 | 14.23% | 43 | 17.48% | 58 | 23.58% | 76 | 30.89% |
| **twolf** | 86594318 | 7434 | 373 | 5.02% | 551 | 7.41% | 725 | 9.75% | 1162 | 15.63% |
| **vortex** | 2698305607 | 16774 | 224 | **1.34%** | 762 | 4.54% | 1650 | 9.84% | 3624 | 21.60% |
| **vpr-place** | 691076933 | 2669 | 213 | 7.98% | 378 | 14.16% | 459 | 17.20% | 590 | 22.11% |
| **vpr-route** | 284712409 | 5542 | 57 | 1.03% | 91 | 1.64% | 180 | 3.25% | 914 | 16.50% |
| **wupwise** | 3992191748 | 1064 | 108 | 10.15% | 183 | 17.20% | 231 | 21.71% | 364 | 34.21% |
| *Average* | *3206857513* | *6562* | *454* | *7.65%* | *852* | *12.50%* | *1209* | *16.68%* | *2181* | *28.44%* |

**Table 2.1: Relationship of instructions and data references.**

number of instructions and the percent of instructions with respect to the total number of different instructions are illustrated in two sub-columns. The values found throughout these benchmarks match the findings of the previous papers [1][26][28]. The results indicate that a disproportionately high number of data references are concentrated in only a small number of instructions. For example, in the 164.gzip benchmark, which contains the largest number of data references, only 73 different instructions (out of 1298) are responsible for generating 75% of the total number of references. This means that 5.62% (found by dividing 73 by 1298) of the total number of instructions that reference memory are causing 75% of the total number of references.

Looking across the table, the percentages of instructions accountable for the data references are very small. The average percentage for generating 90%, 95%, and 99% of the data references are 12.50%, 16.68%, and 28.44%, respectively. For the extreme cases, only 1% of their instructions cause 75% of all the data references (in the vpr-route benchmark, for instance). As we see in Table 2.1, even for 99% of the references, the fraction of instructions causing the most references is less than 0.42 for all benchmarks except the applu benchmark, which has a value of .58. Furthermore, among the various benchmarks, around 10% of the unique instructions on average make more than 90% of the dynamic references. This result indicates that 90% of the execution time is spent on only 10% of the code, which corresponds to the 90/10 locality principle as stated in [1][24][26].

Instead of displaying the information for the data references as in Table 2.1, Table 2.2 depicts how the number of unique instructions is related to cache misses that occur during each simulation in a 16K direct-mapped cache. Since the 90/10 locality principle is valid for data references, a similar trend between the number of instructions and cache misses can also be found. Observed from the data in Table 2.2, we see that the results are as expected, that a very small number of the instructions cause most of the data misses. Among all the benchmarks, four benchmarks (ammp, gzip, vortex, and vpr-route), have fewer than 1% of all the different instructions cause 75% of the cache misses. Also, the percentage of instructions that account for about 75% of all misses is only 3.74% on average. These numeric values confirm the finding of the previous papers showing that a large percentage of data misses are generated by a small number of PC values.

| SPEC2000 | Number of Misses | # of Insts | Percent of Total Misses | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 75% | | 90% | | 95% | | 99% | |
| | | | # of Insts | % of Total | # of Insts | % of Total | # of Insts | % of Total | # of Insts | % of Total |
| **ammp** | 314481977 | 1986 | 10 | 0.50% | 50 | 2.52% | 134 | 6.75% | 494 | 24.87% |
| **applu** | 7994873 | 2443 | 90 | 3.68% | 273 | 11.17% | 548 | 22.43% | 1209 | 49.49% |
| **apsi** | 276897082 | 1147 | 60 | 5.23% | 82 | 7.15% | 108 | 9.42% | 249 | 21.71% |
| **art** | 392139351 | 373 | 12 | 3.22% | 26 | 6.97% | 36 | 9.65% | 62 | 16.62% |
| **crafty** | 82002112 | 2866 | 156 | 5.44% | 296 | 10.33% | 418 | 14.58% | 741 | 25.85% |
| **equake** | 17794232 | 550 | 34 | 6.18% | 69 | 12.55% | 104 | 18.91% | 209 | 38.00% |
| **gcc** | 17607267 | 23420 | 1148 | 4.90% | 2948 | 12.59% | 4722 | 20.16% | 9634 | 41.14% |
| **gzip** | 1225137379 | 529 | 5 | 0.95% | 15 | 2.84% | 30 | 5.67% | 83 | 15.69% |
| **mcf** | 10363355 | 416 | 11 | 2.64% | 29 | 6.97% | 46 | 11.06% | 92 | 22.12% |
| **mesa** | 5188453 | 647 | 10 | 1.55% | 21 | 3.25% | 35 | 5.41% | 62 | 9.58% |
| **mgrid** | 667237847 | 321 | 12 | 3.74% | 35 | 10.90% | 65 | 20.25% | 122 | 38.01% |
| **parser** | 55008784 | 3973 | 82 | 2.06% | 247 | 6.22% | 442 | 11.13% | 1103 | 27.76% |
| **swim** | 23080138 | 226 | 27 | 11.95% | 35 | 15.49% | 49 | 21.68% | 71 | 31.42% |
| **twolf** | 4196327 | 1800 | 79 | 4.39% | 166 | 9.22% | 228 | 12.67% | 382 | 21.22% |
| **vortex** | 100126715 | 5764 | 25 | **0.43%** | 160 | 2.78% | 339 | 5.88% | 910 | 15.79% |
| **Vpr-place** | 25652003 | 740 | 35 | 4.73% | 72 | 9.73% | 105 | 14.19% | 179 | 24.19% |
| **Vpr-route** | 18513778 | 1909 | 13 | 0.68% | 32 | **1.68%** | 54 | **2.83%** | 174 | **9.11%** |
| **wupwise** | 217978065 | 359 | 18 | 5.01% | 29 | 8.08% | 50 | 13.93% | 90 | 25.07% |
| *Average* | *192299985* | *2,748* | *101.5* | *3.74%* | *255* | *7.80%* | *417* | *12.59%* | *881* | *25.42%* |

**Table 2.2: Relationship of instructions and cache misses.**

Comparing the results of Table 2.1 and Table 2.2, we see that the fraction of instructions that account for the largest fraction of misses are almost 50% less than the fraction of instructions that account for data references. Abraham et al. in [1] state that this fact demonstrates that the high number of misses caused by a few instructions is not due simply to the fact that these instructions are making a lot of references. They also point out that due to the concentration of data misses among a few instructions, these misses can be dealt with using manual or compiler techniques such as their selective line scheduling prefetching technique. As mentioned previously, instead of using selective scheduling on data cache prefetching, Tyson et al. in [24][26] decided to perform cache data placement using a variant of selective line scheduling.

## 2.4 Selective Cache Line Replacement

Several selective line replacement algorithms were proposed in [24][26], including using either static or dynamic information. The underlying idea behind these algorithms was to prevent caching the data brought in from troublesome instructions that cause high misses, based on the assumption that this data was being replaced frequently and therefore should not displace data that has a higher probability of being reused.

In order to prohibit the cache from storing unwanted data, the instructions (identified by their PC values) that generate high miss rates will be recorded. As described in [24][26], once these instructions are found, they are marked as CNA (Cacheable/Non-Allocatable). Basically, once they are marked CNA, the cache allocation policy of the hardware management algorithm will not be performed on the data reference generated by these CNA instructions. The goal is to decrease the high miss rate caused by some instructions.

[24][26] explain the reason for the decrease in the miss rate with a scenario. Suppose there are two items, A and B, that both map to the same line in the cache. However, if A and B repeatedly needed to access this same cache line, then they will cause a large number of misses since A will replace B every time (and vice versa). However, if A can be prevented from entering the cache, then B will experience a much lower miss rate.

The algorithm does not forbid A from entering the data cache forever. The data at A is not cached only when it is brought in from an instruction marked CNA. It will enter the cache if it is brought in by a non-CNA instruction. The marked CNA instructions also

only change the allocation mechanism of the data on a miss - the rest of the hardware cache management algorithm will not be affected.

### 2.4.1 Static Method

The first simple static CNA method marks every instruction that causes misses more than 75% of the time as being CNA. (As noted in [24][26], comparing the result produced by a couple other values, 75% gives the most desirable results.) This value allows an adequate number of load references to be removed from the cache to help performance. To evaluate the effectiveness of this strategy, the cache hit rate and memory bandwidth utilization were analyzed. The results of their simulations reveal a slight decrease in the hit rate, but an overall decrease of required bandwidth of almost 30%.

They measured memory activity in addition to miss rate and bandwidth. They determined these memory traffic changes by comparing the number of references obtained before the integrating of the CNA scheme with the number of references after the transformation. They found that by using this scheme, some instructions generated lower miss rates than before. However, the average results indicated that many other instructions suffered higher miss rates after the incorporation of the scheme. Overall, the memory traffic increased by about 29%, which creates a negative impact on the effectiveness of this method.

Consequently, [24][26] concluded that instructions were being marked as CNA too soon, and they came up with an improved version of the static method. The improved version of the static technique attempts to decrease the number of instructions marked

CNA in order to reduce the number of instructions that suffer an increase in miss rate. This method uses the same scheme and same threshold value used in the previous configuration, but instead of permanently marking a PC as CNA, a PC can be removed from the list if it does useful work. The way to determine if this PC does useful work is to evaluate whether it can do helpful prefetching into the cache more than 75% of the time.

An instruction performs useful prefetch when the cache line brought in by this instruction is later referenced by another instruction before it is replaced. In order to clearly identify these instructions, every cache line has associated with it the PC value that brings this cache line into the cache. This modification of the simple static scheme marks fewer instructions as being CNA, and can better distinguish the instructions that do useful work from the ones that do not. Simulations of this approach showed that overall bus bandwidth can be reduced by 30% on average and can be further reduced by over 50% compared to the simple static method, for five of the programs (out of twenty six). The cache performance on average was only a little bit worse than a regular cache without selective line replacement, but compared to the simple static method, the average hit rate went up. Also, Tyson et al. in [24][26] found that the number of instructions having high miss rates decreased significantly. Because of this reduction, the overall memory traffic on average almost matched the memory traffic of a regular cache.

### 2.4.2 Dynamic Method

The improved static method clearly provides a good way to improve cache performance by reducing memory bandwidth while maintaining the same hit rate and

traffic between the main memory and caches. However, one of the major pitfalls of the static method in general is the requirement of pre-execution profiling to obtain information on instructions and cache misses. Dynamic methods can eliminate the need for training runs of the programs, and can collect the needed information on the fly during execution. Unlike the static method, which uses a specific threshold value to determine the CNA instructions, the dynamic method uses a strategy similar to a branch prediction table to identify CNA instructions. The branch prediction table contains the branch history of a branch instruction identified by its address. The prediction of the branch will depend on the branch pattern of this branch, located in the branch prediction table. Similarly, the dynamic CNA method will use a prediction table to associate a PC with its miss rate history to capture reference patterns of the instructions.

For the dynamic 2-bit counter scheme proposed, each PC table entry will be tagged with a 2-bit counter, called the miss prediction counter. This counter's initial state is "00", and its saturated state is "11". Every time the PC causes a miss, the counter value associated with this PC table entry will increase by 1 until the counter reaches its saturated state. On the other hand, the counter value will decrease by 1 once the PC causes a hit until the counter hits "00".

For this scheme, the cache allocation policy will be called on a miss only if the counter is in states "00", "01", or "10". When the counter value is in state "11", it implies that this particular PC has less than 0.25 hit rate and data associated with this PC will not be cached. Once the counter of an instruction reaches its saturated state, this instruction is considered troublesome and will be labeled CNA.

Similar to the improved static method, an instruction can be unmarked as CNA even after it makes it on the CNA list. This strategy can reduce the huge amount of memory traffic generated by the simple static method. However, the overall memory traffic measured is still much higher than for unmodified caches, and to further reduce memory traffic, the dynamic method incorporates the idea used in the improved static method. Instead of decreasing the counter only when the PC causes a hit, the counter of the PC that does useful prefetch will also be decremented by 1. Like the improved static method, a PC bringing a cache line into the cache will be linked to this cache line. Whenever there is a hit, the values of one or two counters may be decreased. Suppose PC A brings in data reference R, and PC B causes a hit when referencing R. In this case the counter values of both PC A and PC B will be decreased. Such a change will make more CNA instructions become non-CNA instructions at the end.

With this new dynamic version, the average hit rate matches the hit rate produced by the improved static method. Moreover, the overall memory activity shows a 1.36% decrease compared to regular caches. Although the memory traffic of the approach is now lower than the improved static method, the number of marked CNA instructions is substantially smaller in this new approach. As a result, the bandwidth requirement is only reduced by about 15%, instead of 30% found using the static method. The improved static method is better at reducing bus activity, but the improved dynamic method does a better job of reducing memory activity and is still able to reduce a significant amount of bandwidth utilization.

Because it has all these advantages, the improved dynamic method is a more attractive choice for the selective line replacement scheme. This improved dynamic

method will be used as a data replacement method in some of the caching schemes I propose and discuss in the next chapter.

# Chapter 3

# Multi-lateral Caches

Most modern processors incorporate a level 1 cache on-chip. This cache is a small fast memory that can improve the machine performance significantly if it is used effectively. To maximize its effectiveness, the cache should be able to manage data cleverly. Multi-lateral caches, for example, have been studied by various researchers, and have demonstrated their ability to improve cache performance [11][15][16][17][25]. Chapter 1 mentions some of the multi-lateral caches that have been proposed, such as the Assist cache, Victim cache, NTS and PCS. Integrating these caching strategies into the data management scheme of an L1 data cache can provide a lower average memory access time.

Multi-lateral caches not only describe smart caches that can recognize useful data, but they also contain two or more different data stores that operate in parallel. A high level view of the multi-lateral cache is depicted in Figure 3.1.ii, and the left side of Figure 3.1.i shows a typical single structure cache. In general, multi-lateral caches can allow data flow between cache A and cache B, like the Assist cache and the Victim cache. However, this thesis will focus on cache models that make use of the EA and PC patterns and prohibit communication between the two caches. Thus, the multi-lateral cache

shown is a more specific type of cache that does not support communication between cache A and cache B.

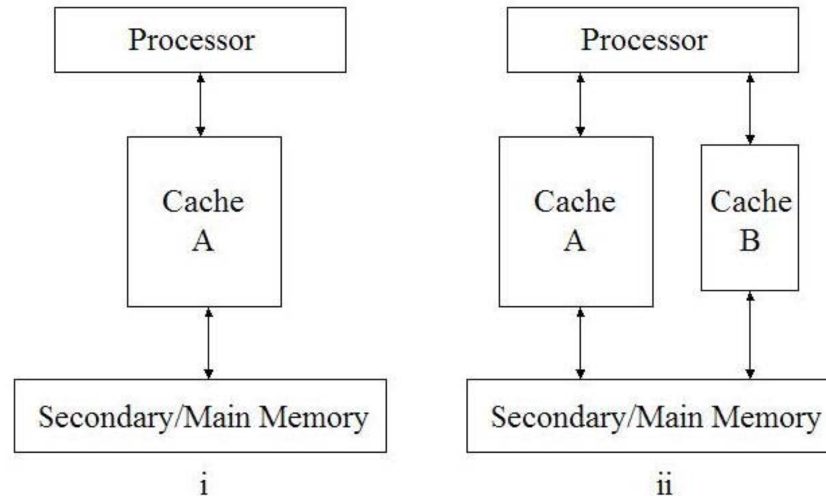


**Figure 3.1: i) Single cache and ii) multi-lateral cache structures [23][25].**

As described previously, multi-lateral caches use smart caching techniques to maximize the storage of useful items in the cache, which will minimize the miss rate and miss penalty. One of these techniques includes the active cache management scheme, which is discussed in [22]. By controlling data block allocation and replacement decisions, this scheme allows caches to select active data to reside in the cache. This differs from standard cache allocation schemes, in which a block of data is passively mapped into its corresponding set based on the associativity of the cache organization [22] without taking advantage of any knowledge of the cache usage characteristics of a block of data. Active cache management, on the other hand, makes data allocation decisions based on the data block usage history, and this helps to determine the usefulness of caching the data.

Similar to cache allocation, caches today do not make use of advanced data replacement algorithms either. This includes multi-lateral caches (the NTS and PCS caches, for example). Although these caches do not make sophisticated decisions regarding block replacement, their cache allocation schemes work well together with block replacements. An excellent allocation scheme benefits block replacement subsequently because the blocks that need to be replaced in the cache are a direct consequence of the data allocation algorithm used on a miss [22].

## 3.1 NTS Data Cache

One of the multi-lateral caches with active block allocation schemes, described in [16][22][25], is the Non-Temporal Streaming (NTS) cache. This cache divides cache blocks into two different groups: temporal (N) and non-temporal (NT), based on the dynamic reuse information gathered as the program runs. The NTS cache bases its block placement decision on the reuse characteristics of the effective address (EA) of the block. A block in L1 is classified as temporal data if any word in the block has been reused during a tour, otherwise, this block is considered non-temporal data. (A tour represents the time period that the block stayed in the cache between its allocation and eviction.) Blocks that show temporal characteristics will be stored in the larger cache (A) and other blocks will be kept in the smaller fully associative cache (B). Assigning temporal data to the larger cache allows data to reside in the cache for a longer time period, since it is likely to be used again in the near future. Non-temporal blocks, on the other hand, are expected to have a shorter life span and therefore get pushed out of the cache more quickly. The small cache B permits fast access to these data.

In addition to the two independent caches working in parallel shown in Figure 3.1.ii, the NTS cache contains a Detection Unit (DU). Figure 3.2 reveals the basic components of this circuit. The DU records and updates the reuse information of the EA, and thus guides data placement for the L1 caches. Whenever there is a memory access, both caches will be checked concurrently for the requested item. Data will be fetched into the processor if found, and no changes will be made to the block's current cache location. If the block is not found (a miss), a decision regarding which L1 cache the incoming block should be put into must be made.

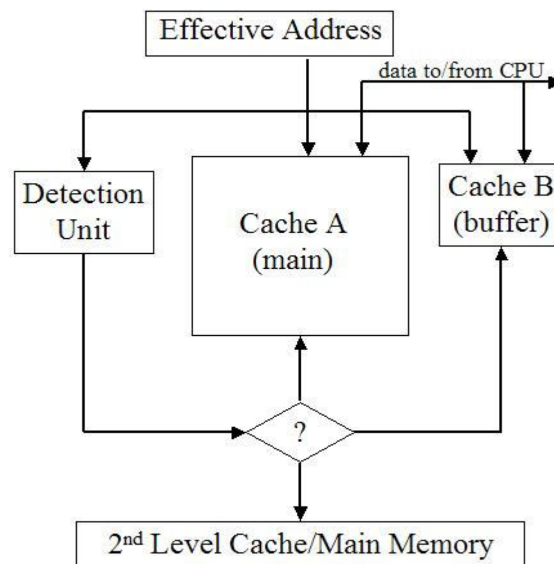


**Figure 3.2: Schematic diagram of NTS cache [22].**

The DU maintains the reuse behavior of the EA associated with the requested block. Each entry in the DU structure consists of one temporal / non-temporal (T/NT) bit and a 32-bit EA. If the EA of the block can be located inside the DU and its reuse behavior is marked with an NT bit, then this block will be loaded in the non-temporal cache, B. On the other hand, if this block's EA entry does not exist in the DU, then a

new entry will be created. By default, this data block will be loaded into cache A and will be categorized as temporal data.

The update of the temporal reuse behavior in the DU happens on the eviction of a block. When a block is evicted from either cache, it first needs to find a matching block address in the DU. If the entry can be found, then the T/NT bit of this entry will be updated accordingly based on its reuse behavior during its just-completed tour in the cache. This entry will be marked T if any word in the block was referenced at least once, and otherwise this entry will be marked NT. If the evicted block cannot find its matching block address in the DU, a new entry for this block's EA will be created and will be marked as temporal. [22][25] found that this implementation of the NTS cache can effectively reduce the miss ratio and improve the overall performance.

Edward et al. in [22] compare a standard 16K single direct-mapped cache model with a 16K NTS model[1]. They found that for the eight SPEC95 benchmarks ran, almost all of the miss ratios are reduced and this finding is depicted in Table 3.1. The miss rates of the gcc, go, and perl benchmarks, for example, were decreased by more than 35%. In particular, the miss rate of the go program has gone down 70%, from 0.070 to 0.021. The only benchmark that shows an increase in miss rate is the swim program, and the increase is less than 1%. The average miss rate across all benchmarks for the NTS cache was reduced substantially compared to a single direct-mapped cache.

As pointed out by Edward et al. in [22], the relative miss ratio alone cannot accurately measure the performance of a cache, because latency masking, miss latency overlap and delayed hits are not taken into account. Consequently, they also compare the speedup of the different caches with respect to their cycle count and the cycle count of

[1] This NTS model contains a direct-mapped 16K cache A and a fully associative 2K cache B.

the base direct-mapped cache. Although the results indicate that the NTS schemes do not fully benefit some programs such as hydro2d, which has less than 1% speedup over the base cache, NTS has improved the cache performance of the compress, gcc, go, and perl benchmarks. The experiments show more than 4%, 3%, 16%, and 5% speedup for these four different programs, respectively.

## 3.2 PCS Data Cache

The Program Counter Selective (PCS) cache, described in [22][25], represents another type of multi-lateral cache that takes advantage of active data management. This cache works similar to the NTS cache, but it utilizes the dynamic program counter value reuse history instead of the effective address reuse characteristics to make data placement decision.

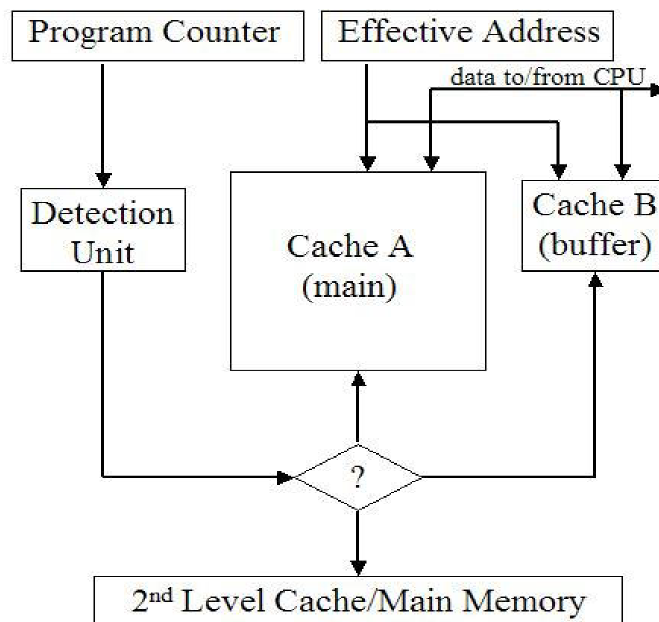


**Figure 3.3: Schematic diagram of PCS cache [22].**

The PCS cache's interaction between the processor, memory and multiple caches is exactly the same as the NTS cache (see Figure 3.1.ii). As with the NTS scheme, there is no direct data path between the two caches. The only difference between the basic structures of PCS and NTS is that the DU of the PCS works with the instructions' program counter instead of the effective address of the reference, as illustrated in Figure 3.3.

Although PCS uses program counter values in its DU, the way PCS operates is almost identical to the NTS cache. During a cache miss, data entering the cache will be allotted to either the larger temporal cache or the non-temporal cache depending on the information stored in the DU. The program counter of the instruction that generated the request for this data block will be searched for in the DU. If no hit occurs, a new entry will be created in the DU, and this data block will be placed into the temporal cache. For a PC match found in the DU, the temporal bit of this PC will determine the allocation of the data. The data will be assigned to cache A if the temporal bit of this PC is set, otherwise to cache B.

When a block is replaced in the cache, based on the reuse characteristics of this block, the temporal bit of its instruction program counter will be set in the DU. Again, a block is considered non-temporal if no word of this data block has been referenced more than once before it falls out of the cache.

[22][25] evaluate the cache performance of this cache as compared to a simple direct-mapped cache and the NTS scheme. Table 3.1 shows the benchmark miss ratios of the NTS, PCS, and the direct-mapped cache for the experiment. Although the PCS causes the lowest hit rate for the hydro2d program among all caches, the overall results of

the PCS scheme demonstrate a significant amount of miss rate reduction when compared to the direct-mapped cache. To be more specific, the miss rates of the go and perl programs have decreased by 50% on average.

Even though the data in the table shows that the miss ratios for most benchmarks are the lowest for the NTS scheme, the swim benchmark's hit rate in the PCS was higher than the NTS hit rate. A similar trend can be found for the performance speedup of these caches where the NTS cache has a greater overall speedup compared to direct-mapped cache than the PCS cache does.

| | compress | gcc | go | hydro2d | li | perl | su2cor | swim |
|---|---|---|---|---|---|---|---|---|
| **NTS** | **0.219** | **0.070** | **0.021** | **0.474** | **0.043** | **0.032** | **0.257** | 0.230 |
| **PCS** | 0.221 | 0.086 | 0.024 | 0.494 | 0.050 | 0.033 | **0.257** | **0.167** |
| **16K** | 0.239 | 0.112 | 0.070 | 0.484 | 0.062 | 0.062 | 0.269 | 0.228 |

**Table 3.1: Miss ratio of the 3 caching schemes running the 8 SPEC95 benchmarks [22].**

## 3.3 Proposed Models - Hybrid Cache Models

It is obvious that the NTS model and PCS model discussed in the previous sections can effectively enhance cache performance by reducing the miss ratio and memory cycle counts. Furthermore, the CNA replacement scheme can save memory bandwidth, which is essential for memory performance measures. Therefore, building caches that use a combination of these techniques might lead to additional performance improvement. The SPEC2000 results presented in Chapter 2 show that a small number of instructions still have a large effect on cache misses and data references, even on today's programs. This suggests that there is still a reason to believe that using the CNA algorithm might help improve cache performance.

Because the dynamic approach of the CNA replacement algorithm does not require training runs of the programs and seems more flexible than the static method, the hybrid caches that I will explore that involve the use of the CNA algorithm will utilize the improved dynamic method. To prevent marking CNA instructions too aggressively, I altered the CNA algorithm to just examine instructions that cause read misses instead of all misses (since read data references usually accounts for 75% of all data references).

The first hybrid cache proposed, NTSCNA (Non-Temporal Streaming and Cachable/Non-Allocatable), will combine the NTS scheme with the CNA algorithm with the goal of maintaining the NTS high hit ratio while also reducing memory bandwidth. The second hybrid cache model, NTSPCS, will utilize both the reuse history of the EA and the PC, with the intent to reduce the miss rate even more. The third hybrid cache, NPCNA, combines all three approaches, attempting to take advantage of the CNA, NTS, and PCS schemes.

### 3.3.1 NTSCNA Data Cache

The NTSCNA cache model (see Figure 3.4), an integration of the CNA algorithm into the NTS data cache, uses both the PC and the EA to determine the caching decision. The CNA scheme is implemented using a 256-entry miss prediction table. As stated in [24][26], a miss table of 256 entries can reduce the average memory bandwidth to a value almost matching the value of an infinite-sized table (for SPEC92 benchmarks). Each entry in this PC history table contains a 32-bit PC value and a counter associated with this particular PC. The LRU (least recently used) algorithm will be used to manage the replacement of the PC entries in the table.

Using the improved dynamic CNA scheme, a PC in the history table will be updated on every cache miss for each memory access. If the PC does not exist in the table, this PC will be added to the table with a default counter value of 0. If the PC exists, its counter value will be checked to determine whether it is marked CNA and it will be updated accordingly. This PC is marked CNA if its counter value has reached the maximum PC counter value of 4. Once a PC is marked CNA, the cache line associated with it will not be cached. On the other hand, if the PC is not yet marked CNA, then the counter value of this PC will be increased by one. For each cache hit, the counter of the PC that causes the hit will be decremented by one. Moreover, the counter of the PC that brought in this cache line originally will also be reduced by one (stopping zero). Lowering the values of both PC counters will help prevent a large increase in memory references due to marking an instruction CNA too fast [26]. As a result, the state of the PC could change from CNA to non-CNA over time, depending on the pattern of cache hits and misses.

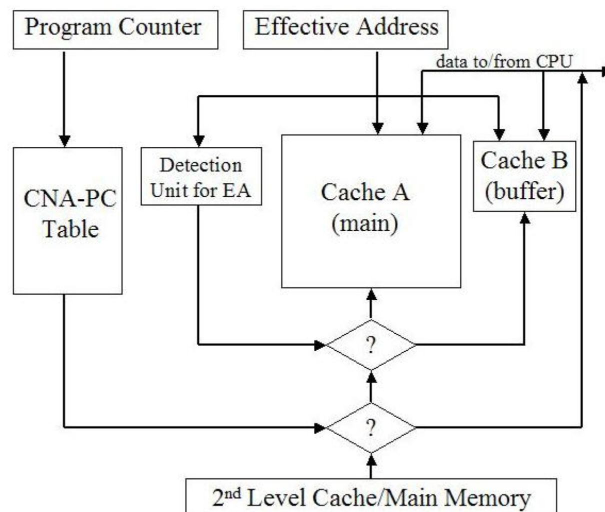


**Figure 3.4: Schematic diagram for NTSCNA cache.**

For all write misses and other non-CNA read misses, the NTS caching scheme will be used to determine whether to store data into temporal cache A or non-temporal cache B, depending on each cache line's effective address. On a miss, the EA that causes the miss will be checked against the EA entries in the DU. As discussed earlier, for each block eviction from the L1 structure, the block's temporal information will be stored in the DU. The T/NT bit in the DU will be set if the cache line showed temporal reuse over the time it resided in the L1 cache. When a cache block is about to enter the level one cache during a cache miss, the DU will first be checked for the EA of the block. If a match is found in the DU, the cache block will be stored in cache A if its EA's temporal bit is 1, otherwise it will be stored in cache B. Once a block is identified as NT and stored in cache B, it will remain in cache B until it is evicted regardless of whether or not that block shows temporal behavior during its stay in the cache.

A new block will be assigned to the temporal cache if its EA does not exist in the DU. A new entry in the DU will be created, and this block will be identified as temporal. For all the experiments conducted, a 32-entry fully associative DU is used. The LRU replacement algorithm is used in the DU, so the entries that have not been used for a long time will be thrown out of the structure.

### 3.3.2 NTSPCS Data Cache

The NTSPCS cache combines both the NTS and PCS methods, and places data in cache according to the existence of the EA and the PC in their DUs and will try to take advantage of the reuse information of both of them. For this structure, two DUs will be used, one to store the EA and the other one for the PC. This is illustrated in **Figure 3.5**. In

addition, a 2-bit counter (NP counter) with maximum value of 3 and initial value of 0 will be used in order to choose which scheme to use to make the caching decision. This approach works similar to a tournament branch predictor, where the method with a better prediction will be used.

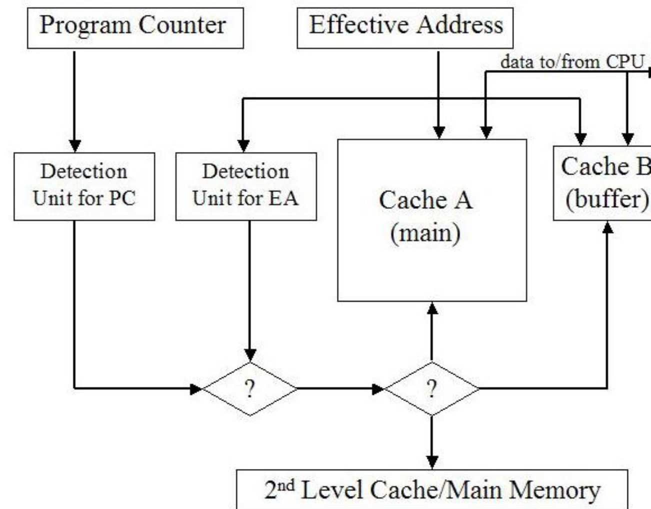


**Figure 3.5: Schematic diagram of NTSPCS cache.**

For both the NTS and PCS schemes, data placement decisions are made on each cache miss. On a miss, the PC and the EA will be searched for in their respective DUs. If neither one is found, the data will be placed in the temporal cache (A) by default, and the NP counter value will not be updated. Both the EA and PC will be added as a new entry to their DUs respectively. If the EA exists in its DU structure and the PC does not, then the NTS method will be used. This means that the EA will be used to determine whether or not to store the data in the temporal or non-temporal cache. The DU for the PC will be updated accordingly by adding a new PC entry, and the NP counter value will be decreased by one.

Likewise, when the PC exists in its DU and the EA does not, a similar approach is used. For this case, the PC is used to make the data placement decision, but the counter value will be increased by 1 instead of decreased.

Finally, if both the EA and PC exist in their DUs, the data will be placed in cache B if both the EA and PC's temporal bits are not set, and in cache A if they show temporal characteristics. If their temporal characteristics disagree (the EA is marked non-temporal and PC is marked temporal, or vise versa), then there are two possible assignments depending on the NP counter value. For a counter value of less than half of the maximum counter size, the NTS method will be used for making the decision. Otherwise, the PCS approach will be used.

For the same NTSPCS cache structure, a different way of determining whether to use the EA or PC is also implemented. Instead of using the counter values to determine whether to use the EA or the PC, the decision is based on which method was previously correct. This NTSPCS-Prev approach is developed hoping to determine its potential for choosing a better or more correct method for data cache placement as compared to the original NTSPCS.

The idea of this scheme is to have its decision based on recent history. Each cache line in L1 will contain an extra EA/PC bit to record whether an entry was brought in to the cache by the NTS or the PCS scheme during each cache miss. Another bit, called the Previous Method bit, contains the information regarding the placement method used when there is a cache hit. On each hit, the Previous Method bit will be set to 1 (for PC) if the cache's EA/PC bit is set to PC with a value of 1, otherwise this bit is set to 0. For each miss, if both EA and PC are not found in their DU, the new data will be placed

in cache A. For this new cache entry, the EA/PC bit for cache A will not be set, which indicates that this cache line is brought in by the EA. When the EA exists in its DU and the PC does not exist, or vise versa, the idea is the same as the above NTSPCS method.

NTSPCS-Prev differs from the simple NTSPCS in that there is no need to update the counter for this scheme. If both the EA and the PC exist, and the EA and PC temporal bit disagree, the EA will be used if the Previous Method bit is not set, otherwise, the PC will be used for determining the data placement.

### 3.3.3 NPCNA Data Cache

This NPCNA cache design, shown in Figure 3.6, combines the NTSPCS and CNA methods and aims to increase the number of cache hits while reducing the memory bandwidth. It not only uses both the EA and the PC to assign data into temporal or non-temporal caches, but also uses the program counter of the instruction to decide whether or not to allocate data into the cache at all.

The NPCNA method operates in a very similar manner to the NTSCNA method discussed previously. On a cache miss, the PC will be used to determine whether or not to cache the data depending on the PC's CNA counter status. Data will not be cached if the counter of the PC that brought in this cache line is in its saturated state, which indicates that the PC is marked CNA. For all non-CNA misses, NPCNA uses the NTSPCS methods previously described to deal with data placement. For each miss event, the DUs contain the EA and PC will be searched for matches. If both searches are successful, data placement will be based on the status of the NP counter. If the counter value is less than half of its counter size, the EA will be utilized for determining data

placement, otherwise the PC will be used. When only one of the searches returns a match, the corresponding NTS or PCS scheme will be utilized and the counter will be updated appropriately. If neither the EA nor the PC is found, the default assignment of the data is to the larger temporal cache.

The NPCNA method was also implemented using the Previous Method bit to make decisions between using the NTS or PCS schemes as similar to the second method of the NTSPCS model as possible. This modification of the NPCNA scheme is called NPCNA-Prev.

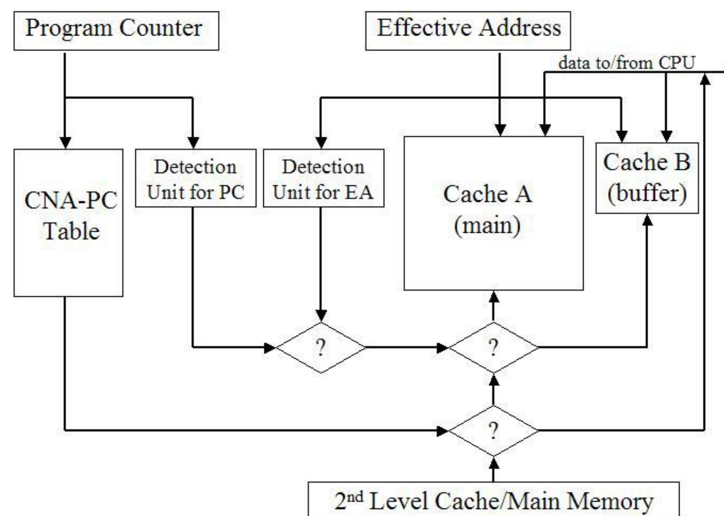


**Figure 3.6: Schematic diagram for NPCNA cache.**

## 3.4 Simulation Environment

The experiments conducted included simulations of the following cache models: single, NTS, PCS, NTSCNA, NTSPCS, NTSPCS-Prev, NPCNA, and NPCNA-Prev. The result of the single, NTS, and PCS models will be compared with the results of the hybrid caches. These simulations involve the use of the Intel ATOM tool and the

mlcache tool set. Intel ATOM simulates the execution of the instructions in a program and produce traces of the program's execution. For this set of experiments, the Intel ATOM tool was configured to produce program traces of data references and program counter values only, and this information was fed into the multi-lateral cache simulator, mlcache.

mlcache [25] is an event-driven and timing-sensitive cache simulator. This tool is based on the Latency Effects (LE) cache timing model and can be easily integrated into other tools such as Intel ATOM. This cache simulator is a highly configurable tool where all routines that specify actions taken in the cache reside in the config.c file. Changing the way the cache works often requires changing only a single file. In addition, all cache parameters such as cache size and latencies are read from a parameter file, and the name of the file is provided to mlcache as a command line argument. In the simulations, the L1 multi-lateral caches contain an 8 KB direct-mapped cache A and a 1 KB fully associative cache B. The small 8 KB cache is used to highlight the effectiveness of the cache schemes simulated. Each cache is configured to have a 32-byte block size and uses the LRU replacement policy. To simulate a conventional cache, the B cache is simply ignored.

A complete list of the multi-lateral cache configurations can be found in Appendix A. All the caches simulated will use this set of parameters consistently, except for the larger single cache that uses a 16 KB cache. The set of benchmarks used will be the set of SPEC2000 benchmarks described in Chapter 2. This subset of the benchmark suite includes eight integer and nine floating-point programs. The benchmarks are run using SPEC2000 test data sets and all of the programs are run to completion. The results

of the simulations of these benchmarks using different cache configurations will be presented in the next chapter.

# Chapter 4

# Results

Multi-lateral caches using active data management in general can improve cache performance significantly. In particular, the NTS and PCS caches that use the reuse characteristics of the EA and PC, respectively, do a fine job of improving cache performance. In addition, the CNA algorithm can maintain cache hit rates yet effectively reduce memory bandwidth, which is important for processor designs today. This chapter will evaluate the performance of the proposed caches, which combine the methods used in NTS, PCS, and the CNA schemes.

## 4.1 Miss Ratio

Although the miss ratio alone is not a good metric for evaluating cache performance (which is pointed out in [22][25]), it does affect the average memory access time directly and serves as a good cache performance indicator. Using the simulation environment discussed in Chapter 3, Table 4.1 shows a detailed breakdown of the miss rates for each benchmark run for the single 8K direct-mapped, NTS, PCS, NTSCNA, NTSPCS, NTSPCS-Prev, NPCNA, NPCNA-Prev, and single 16K direct-mapped caches. The table shows there is no consistent pattern for the benchmarks' miss rates because the

pattern of each set of miss rates for individual benchmarks varies slightly. For example, for the art benchmark, the single 8K cache causes fewer cache misses than the NTS, NTSCNA, NPCNA, and the NPCNA-Prev caches. NTSCNA can be seen to have a high miss rate of 0.59, higher than the 0.524 for a single 8K cache. For this program, enlarging the single cache to 16K does not seem to help too much, but the PCS, NTSPCS, and NTSPCS-Prev all exhibit a miss ratio that is either lower than or almost matches the miss ratio of the single 16K cache. Such results indicate that using reuse characteristics of the PC for cache management helps reduce some misses which cannot be eliminated using either the EA schemes or a larger cache.

A similar pattern applies to the swim program, where schemes using the PC reuse information produce fewer misses than all but the NPCNA and NPCNA-Prev approaches. However, the wupwise benchmark does not do well utilizing the PCS caching scheme - its miss rate went from 0.119 (single 8K) up to 0.166, whereas using the NTS scheme caused the miss rate to drop to 0.109. Unlike the art, swim, and wupwise benchmarks, the vortex program runs better on all of the multi-lateral caches, even the NPCNA and the NPCNA-Prev schemes which generally have the highest number of misses. The numbers of hits for all the multi-lateral caches for this program even outnumber the hits for the single 16K cache. Figure 4.1 provides the average miss ratios of all the benchmarks ran for all cache configurations and these averages will give a general idea of their cache performance.

Compared to the single 8K direct-mapped cache, the NTS and PCS caches, as expected, show a significant amount of cache miss reduction and match the results of the studies done in [22][23][25]. For all other cache configurations, they show better hit

rates when compared to the single 8K cache with the exception of the NPCNA-Prev cache, which causes more misses. Even though the NTSCNA generates fewer misses than the single 8K cache, it does not maintain its average miss rate too well when compared to its base model, the NTS cache. One possible explanation can be that CNA algorithm interferes with the reuse characteristics of the EA too often. With the CNA scheme, data blocks are not stored in the cache when the instruction that causes the data block to be brought in is marked CNA. Consequently, some non-cached data blocks that might show a temporal trait for the EA might not be recorded. Also, a high number of instructions marked CNA can negatively affect the number of misses.

Generally, the probability that data in consecutive locations of a block will be referenced is often quite high. Not caching the block associated with a CNA instruction can lead to misses for other data nearby in the block.

Another model, NTSPCS, takes advantage of both the PC and EA reuse information, and has the lowest miss rate among all multi-lateral caches. Using more information might lead to a more accurate cache allocation for the data block, which will help to increase the number of cache hits.

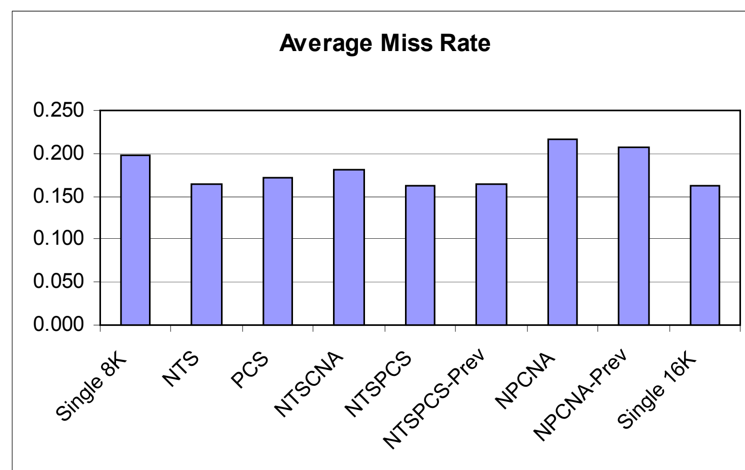


**Figure 4.1: Average miss rates for all simulated caches.**

| SPEC2000 | **Single 8K** | **NTS** | **PCS** | **NTSCNA** | **NTSPCS** | **NTSPCS-Prev** | **NPCNA** | **NPCNA-Prev** | **Single 16K** |
|---|---|---|---|---|---|---|---|---|---|
| **ammp** | 0.234 | **0.214** | 0.226 | 0.229 | 0.216 | 0.217 | 0.290 | 0.285 | 0.215 |
| **applu** | 0.127 | **0.086** | 0.094 | 0.107 | 0.088 | 0.087 | 0.145 | 0.146 | 0.089 |
| **apsi** | 0.214 | 0.145 | 0.154 | 0.150 | **0.133** | 0.138 | 0.149 | 0.155 | 0.196 |
| **art** | 0.524 | 0.526 | 0.517 | 0.590 | 0.518 | **0.516** | 0.544 | 0.544 | 0.520 |
| **crafty** | 0.184 | 0.083 | 0.121 | 0.080 | 0.082 | 0.084 | 0.081 | 0.082 | **0.071** |
| **equake** | 0.203 | 0.182 | 0.182 | 0.209 | 0.181 | 0.181 | 0.303 | 0.300 | **0.170** |
| **gcc** | 0.128 | 0.081 | 0.089 | 0.083 | **0.080** | **0.080** | 0.093 | 0.093 | **0.080** |
| **gzip** | 0.122 | 0.112 | 0.114 | 0.146 | 0.115 | 0.114 | 0.171 | 0.165 | **0.106** |
| **mcf** | 0.458 | 0.420 | 0.403 | 0.420 | 0.403 | 0.408 | 0.404 | 0.404 | **0.390** |
| **mesa** | 0.056 | 0.048 | 0.051 | 0.048 | 0.049 | 0.049 | 0.056 | 0.056 | **0.043** |
| **mgrid** | 0.103 | 0.081 | 0.080 | 0.102 | 0.079 | 0.079 | 0.420 | 0.292 | **0.073** |
| **parser** | 0.267 | 0.243 | 0.244 | 0.244 | 0.242 | 0.242 | 0.265 | 0.271 | **0.226** |
| **swim** | 0.243 | 0.241 | **0.234** | 0.293 | **0.234** | 0.239 | 0.268 | 0.268 | 0.238 |
| **twolf** | 0.142 | 0.117 | 0.136 | 0.130 | 0.119 | 0.133 | 0.141 | 0.131 | **0.108** |
| **vortex** | 0.093 | **0.039** | 0.041 | 0.041 | 0.040 | 0.042 | 0.054 | 0.052 | 0.069 |
| **vpr-place** | 0.111 | 0.081 | 0.096 | 0.081 | 0.090 | 0.092 | 0.094 | 0.095 | **0.072** |
| **vpr-route** | 0.222 | 0.152 | 0.155 | 0.170 | 0.155 | 0.154 | 0.214 | 0.215 | **0.145** |
| **wupwise** | 0.119 | **0.109** | 0.166 | 0.122 | 0.110 | **0.109** | 0.194 | 0.178 | **0.109** |
| *Average* | *0.197* | *0.164* | *0.172* | *0.180* | *0.163* | *0.165* | *0.216* | *0.207* | ***0.162*** |

**Table 4.1: Miss ratio of all cache configurations.**

The NTSPCS-Prev, on the other hand, shows only a small increase in the number of misses as compared to the NTSPCS and the NTS schemes, but it still has an advantage over the PCS cache. Using the Previous Method bit does not seem to reduce the number of misses over the simple NTSPCS scheme. However, the advantage of the NTSPCS scheme does not get carried over to the NPCNA caches. The miss rate increased substantially for both the NPCNA and NPCNA-Prev caches, which is much worse than for the NTSCNA cache. The two reasons discussed previously about poor miss rates for the NTSCNA scheme apply to these two schemes as well.

Finally, the 16K direct-mapped cache has the lowest miss ratio among all cache configurations. This is not surprising, since large caches can reduce both capacity and

conflict misses, the two major types of cache misses. (It is worth noting, however, that the NTSPCS miss ratio is within 0.001 of the 16K cache.)

## 4.2 Cache Speedup

When memory latencies are taken into account, cache speedup can assess cache performance more accurately. I measured the total number of cycles for reads and writes in the L1 cache, where the latency for an L1 hit is 1 and the latency for a miss is 18[2] (see Appendix A for complete configuration). The speedup of the caches will be calculated relative to the single 8K direct-mapped cache model. (I calculate the speedup fraction by dividing the total number of cycles for the single 8K cache by the total number of cycles for the new cache, subtract one, and then multiply by 100%.)

Table 4.2 summarizes the speedup of all the cache configurations for all benchmarks. The speedup fractions exhibit similar characteristics as the miss ratios discussed in the previous section. The table illustrates that not only are there some programs that do not benefit from the proposed models, but also that they require many more cycles than the base model. In the NTSCNA, NPCNA, and NPCNA-Prev caches, for example, many benchmarks produce negative speedups. For the swim program, the NTSCNA configuration is 26 percent slower and is the slowest cache for this benchmark among all other configurations. However, its average speedup still shows a net gain for this cache compared to the single 8K cache, and it has the advantage of saving bandwidth (although its speedup is much less than the NTS performance).

---

[2] 18 cycles of latency was used so that the results can be easily compared with other results on the NTS and PCS schemes of similar works, [17] and [22].

For the NPCNA and NPCNA-Prev caches, the equake, mcf, mgrid, and wupwise benchmarks produce 40% higher cycle counts than the single 8K model. For all programs run, the NPCNA and NPCNA-Prev caches, having negative speedup on average, do not seem to be too promising. The NTSPCS and NTSPCS-Prev show the best results. NTSPCS's crafty test run outperforms all caches except for the single 16K. Also, the NTSPCS-Prev has a marginal speedup compared to all other caches, including the single 16K cache, for the gcc program simulations. For the majority of the benchmarks ran, the NTSPCS-Prev cache does not reduce as many simulation cycles as the simple NTSPCS scheme.

| SPEC2000 | **NTS** | **PCS** | **NTSCNA** | **NTSPCS** | **NTSPCS-Prev** | **NPCNA** | **NPCNA-Prev** | **Single 16K** |
|---|---|---|---|---|---|---|---|---|
| **ammp** | **8.64%** | 3.43% | -2.72% | 7.09% | 6.81% | -28.00% | -26.71% | 7.28% |
| **applu** | **45.44%** | 34.26% | 8.91% | 42.47% | 44.23% | -22.52% | -22.59% | 42.24% |
| **apsi** | 45.24% | 39.10% | 26.25% | **62.16%** | 56.07% | 23.10% | 18.26% | 8.05% |
| **art** | -0.34% | 0.89% | -18.37% | 0.91% | **1.35%** | -12.51% | -12.67% | 0.74% |
| **crafty** | 108.79% | 48.16% | 115.59% | 130.37% | 108.34% | 108.37% | 106.35% | **141.15%** |
| **equake** | 12.91% | 12.89% | -19.02% | 13.55% | 13.57% | -46.07% | -45.52% | **18.74%** |
| **gcc** | **58.95%** | 44.81% | 47.35% | 59.20% | 59.55% | 23.29% | 23.18% | 56.94% |
| **gzip** | 7.50% | 6.71% | -24.83% | 5.37% | 6.86% | -38.37% | -35.99% | **15.49%** |
| **mcf** | 7.27% | 11.35% | -0.02% | 11.35% | 9.90% | -44.08% | -44.13% | **15.54%** |
| **mesa** | 20.40% | 13.21% | 17.84% | 17.36% | 16.37% | -12.07% | -13.25% | **32.01%** |
| **mgrid** | 27.39% | 29.41% | -6.76% | 30.23% | 30.22% | -78.54% | -69.07% | **40.41%** |
| **parser** | 10.76% | 9.77% | -7.05% | 10.74% | 10.75% | -17.52% | -18.79% | **16.42%** |
| **swim** | 0.98% | **3.66%** | -25.94% | **3.66%** | 2.23% | -17.69% | -17.69% | 2.12% |
| **twolf** | 22.61% | 6.90% | 8.87% | 19.69% | 8.35% | -3.29% | 4.56% | **34.17%** |
| **vortex** | **137.35%** | 126.81% | 116.13% | 132.71% | 119.68% | 57.39% | 62.87% | 34.80% |
| **vpr-place** | 37.41% | 17.20% | 31.41% | 23.33% | 19.49% | 5.35% | 4.49% | **52.40%** |
| **vpr-route** | 45.32% | 42.60% | 18.76% | 42.42% | 43.20% | -11.00% | -11.66% | **52.57%** |
| **wupwise** | 9.40% | -24.67% | -16.58% | 8.56% | 9.90% | -52.20% | -47.79% | **11.48%** |
| *Average* | *33.67%* | *23.69%* | *14.99%* | ***34.51%*** | *31.49%* | *-9.24%* | *-8.12%* | *32.36%* |

**Table 4.2: Percentage of speedup for all caches over the single 8K cache.**

The NTSPCS cache performs better than the NTS scheme according to the average miss ratio shown in Table 4.2. However, unlike the miss ratio data, the scheme

that shows the greatest speedup over the single 8K cache is the NTS. The number of total cycles counted is a more precise measurement of cache performance and does not correspond to miss ratios sometimes. The total number of misses refers to the number of cache accesses when the requested data cannot be found during the time of access, and includes delayed hits.

According to [23], a delayed hit occurs when the processor makes two requests to the same data block at cycles X (first request) and Y (second request), and both requests are misses. In order to have a delayed hit, the number of cycles between time X and Y needs to be less than the specified memory latency, and the cache must be lockup-free. As a result, Y completes its request after cycle X but less than the number of memory latency cycles. Thus, delayed hits will experience less than the nominal hit latency. For two caches with similar miss ratios, the total number of simulated cycles can be less if there exists a large number of delayed hits. For example, the simulated total number of cycles for NTSPCS is less than the number of cycles for the NTSCNA scheme even though the NTSPCS has a higher miss ratio. The same pattern can be found for the art and vpr-route programs for the PCS and NTSPCS schemes and PCS and NTSPCS schemes respectively. Based on the total number of cycles for the benchmarks using different multi-lateral caches, the NTSPCS scheme provides the greatest speedup.

## 4.3 Memory Bandwidth

Memory activity is an important metric for determining the effectiveness of a caching scheme, since memory traffic has a large impact on processor performance. Eliminating the overhead generated by transferring large blocks of data can help to

minimize memory traffic. Using the CNA algorithm can reduce the required memory bandwidth, and the memory bandwidth saved will be proportional to the number of references that are made to the marked CNA instructions. For a CNA miss, only one word (4 bytes in this case) will be fetched from memory instead of the whole cache line (32 bytes), since no data needs to be cached. As a result, the more CNA references there are, the more memory bandwidth can be reduced. To calculate the total amount of memory bandwidth saved for each benchmark, I will use the following equation (3):

$$\textit{Saved Memory Bandwidth} = \textit{\# of references to marked CNA instruction} * (\textit{block size} - \textit{word size}). \quad (3)$$

| SPEC2000 | **NTSCNA** | **NPCNA** | **NPCNA-Prev** |
|---|---|---|---|
| **ammp** | 9,765,428,708 | **16,092,270,208** | 15,837,899,784 |
| **applu** | 193,523,708 | **507,649,940** | 507,073,000 |
| **apsi** | 5,977,699,980 | 7,460,313,028 | 7,460,313,028 |
| **art** | **14,313,896,240** | 13,651,781,780 | 13,704,322,324 |
| **crafty** | 172,552,884 | **752,642,324** | 743,098,608 |
| **equake** | 1,136,743,076 | **2,138,713,892** | 2,108,117,536 |
| **gcc** | 243,200,048 | **464,904,944** | 464,248,260 |
| **gzip** | 51,377,129,244 | **89,759,354,216** | 85,544,294,136 |
| **mcf** | 403,102,140 | **450,512,916** | 450,032,772 |
| **mesa** | 67,267,116 | 257,115,740 | **268,150,232** |
| **mgrid** | 14,355,307,232 | **149,620,882,012** | 89,759,354,216 |
| **parser** | 2,430,594,040 | **4,025,862,176** | 4,015,026,960 |
| **swim** | **418,755,904** | 313,801,572 | 313,720,008 |
| **twolf** | 32,843,244 | **68,066,712** | 64,635,032 |
| **vortex** | 566,505,296 | **1,945,042,316** | 1,744,148,420 |
| **vpr-place** | 282,984,184 | **797,979,336** | 769,359,640 |
| **vpr-route** | 612,604,356 | 1,149,786,092 | **1,166,775,372** |
| **wupwise** | 7,279,483,288 | **17,150,771,400** | 15,825,848,836 |
| *Average* | *6,090,534,483* | ***17,033,747,256*** | *13,374,801,009* |

**Table 4.3: Memory bandwidth saved in bytes for NTSCNA, NPCNA, and NPCNA-Prev caches.**

This equation (3) overestimates the memory bandwidth saved compared to a cache without the integration of the CNA algorithm. If a CNA instruction did not bring in the requested data block into the cache, and a word in this data block is referenced later

on, then there is an increase in memory bandwidth needed. The second reference would have been a hit using a normal cache replacement algorithm and the memory bandwidth used previously will be wasted. This formula did not account for this type of memory bandwidth loss, but this information can be found by adding together the number of times a CNA instruction becomes non-CNA, and multiplying by that line size. Due to time constraints, this loss in bandwidth was not determined; however, this loss will not significantly affect the memory bandwidth saved calculated from equation (3).

The number of memory bandwidth bytes saved will be evaluated for the three caches that incorporate the CNA algorithm, namely the NTSCNA, NPCNA, and NPCNA-Prev caches. Table 4.3 illustrates the calculated results of the simulations ran based on equation (3). Each of these schemes can reduce an enormous amount of memory bandwidth compared to the base cache models. These numbers reveal that the NPCNA scheme can reduce memory bandwidth the most, followed by NPCNA-Prev and NTSCNA (with the exception of the art benchmark).

## 4.4 Counter Configuration Impacts

Since the miss ratio and speedups for the proposed methods are not able to outperform (or even match) the performance of the base NTS cache, I experimented with fine-tuning the counters used in these caches to see if I could improve their performance. The next set of experiments run involves caches using counters (NTSCNA, NTSPCS, NPCNA, and NPCNA-Prev). The size of the counter corresponding to each PC in the history table and the counter’s decrement values will be modified for the NTSCNA scheme. For the NTSPCS scheme, the same adjustments will be applied to the NP

counter, which is used to determine whether to use the EA or PC to do data allocation. Moreover, both the increment and decrement values will be altered. Depending on the results of the NTSCNA and NTSPCS experiments, the counter(s) of the NPCNA and NPCNA-Prev caches will be changed accordingly.

### 4.4.1 NTSCNA Changes

There are a total of five different counter configurations, where the maximum counter value is set to 4 and 8. For each hit in the cache, the counter of the PC that causes the hit and the counter of the PC that brings in the cache line will be decreased with a value that is one or more. This modification is to prevent the CNA approach from marking instructions CNA too often. To reduce showing the tremendous amount of data generated by the different configurations, Figure 4.2 graphs the average miss ratio whereas Table 4.4 shows the simulated cycles, number of references to CNA instructions, and memory bandwidth savings for all the NTSCNA cache configurations.

Using a counter size of eight with decrement value of one yields the best result among the various choices. It can reduce miss rates substantially (to 0.1671), which almost matches the average miss rate of the NTS scheme of 0.1645. At the same time, it saves a large amount of memory bandwidth by labeling some instructions as CNA. The 8d4 (counter size of 8, decrement by 4) configuration does not work as well, because too few instructions are marked CNA. Due to the low bandwidth savings, 8d4 does not seem to be the best choice although it does require the fewest number of cycles.

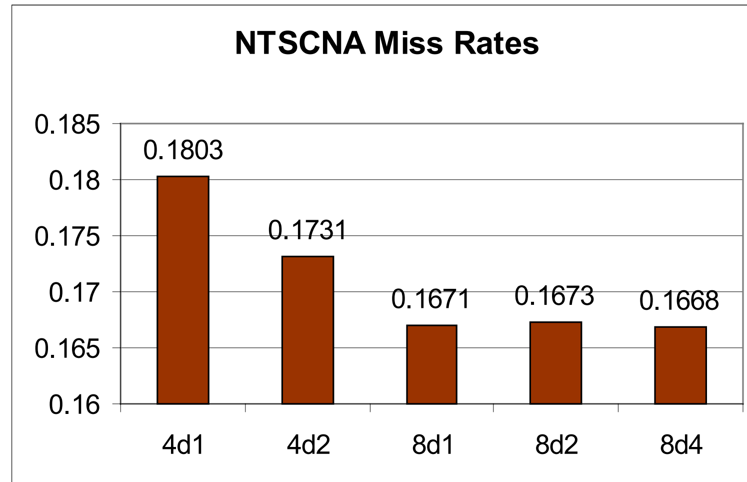


**Figure 4.2: Average miss rates of NTSCNA with different counter configurations.**

| Configuration | Cycle Count | Speedup | # of Reference to CNA | Bandwidth Savings[3] (bytes) |
|---|---|---|---|---|
| **4d1** | 7,225,624,575 | 0.1499 | **217,519,089** | **6,090,534,483** |
| **4d2** | 6,313,391,865 | 0.2147 | 143,261,266 | 4,011,315,440 |
| **8d1** | 6,228,531,669 | 0.2723 | 123,242,179 | 3,450,781,006 |
| **8d2** | 5,739,068,660 | 0.2760 | 89,565,475 | 2,507,833,295 |
| **8d4** | **5,712,068,782** | **0.2898** | 78,690,131 | 2,203,323,673 |

**Table 4.4: Average experiment results for NTSCNA with different counter configurations.**

### 4.4.2 NTSPCS Changes

In addition to the default configuration, there will be three more counter configurations modeled in this experiment. Two counter sizes of 4 and 8 will be used for the NP counter in the NTSPCS cache, and the update value of this counter will vary. For example, the counter configuration of 4i2d1 represents a counter of size four which is increased by 2 when the PC of the block evicted finds its match in its DU but the EA could not find its match in its DU, and decreased by one for the opposite case. The 4i1d2 and 4i2d1 settings are biased toward the NTS or PCS scheme correspondingly. The

[3] Memory bandwidth savings are calculated using the equation (3).

8i1d1 setting holds a longer history pattern of the EA and PC existences in their DUs, with the goal of providing better prediction between using the NTS or PCS strategy.

| Configurations | **Miss Rate** | **Cycle Count** | **Speedup** |
|---|---|---|---|
| **4i1d1** | **0.1630** | 5,485,429,400 | 0.1867 |
| **4i2d1** | 0.1639 | 5,505,756,367 | 0.1823 |
| **4i1d2** | 0.1640 | **5,457,408,957** | **0.1928** |
| **8i1d1** | 0.1635 | 5,466,430,803 | 0.1908 |

**Table 4.5: Average experiment results for NTSPCS with different configurations.**

Table 4.5 shows the average miss rate, the total cycle count, and amount of speedup relative to the single 8K direct-mapped cache. The findings reflect that there does not appear to be much advantage to changing the configuration of the NP counter, since the 4i1d1 (default) NP counter yields the lowest miss rate and relatively low cycle counts. This cache hardly benefits from having a counter that is biased toward using the EA or PC, and having longer histories of EA and PC usage pattern (like the 8i1d1) does not seem to help the cache performance either.

### 4.4.3 NPCNA and NPCNA-Prev Changes

Since there was no improvement for changing the NP counter configurations, the set of experiments I ran for the NPCNA will only alter the configuration of the PC counter for the CNA algorithm. To be consistent, NPCNA and NPCNA-Prev will use the same counter settings as the NTSCNA experiments: 4d1 (default), 4d2, 8d1, 8d2, and 8d4. Examining the results in Table 4.6, the performance of caches does improve when the counter size and the decrement values increase. The average number of cycle counts has decreased almost 50% between the 8d4 and 4d1 configurations for both schemes. While the performance of the caches improved for these settings, the bandwidth savings for both schemes have dropped significantly (by more than 50%). These numbers imply

that these two caches suffer from marking the instructions too aggressively. Removing some of the CNA instructions that do useful prefetches is the key to obtaining higher performance gains.

| Configurations | *NPCNA* | | | | |
|---|---|---|---|---|---|
| | **Miss Rate** | **Cycle Count** | **Speedup** | **Ref to CNA** | **Bandwidth Savings** |
| **4d1** | 0.2159 | 12,892,730,542 | -0.4951 | **608,348,116** | **17,033,747,256** |
| **4d2** | 0.2068 | 11,736,943,269 | -0.4454 | 537,581,879 | 15,052,292,601 |
| **8d1** | 0.1879 | 7,591,149,298 | -0.1425 | 285,286,699 | 7,988,027,564 |
| **8d2** | 0.1762 | 7,121,199,984 | -0.0859 | 237,712,660 | 6,655,954,468 |
| **8d4** | **0.1703** | **6,461,099,379** | **0.0075** | 192,152,374 | 5,380,266,464 |
| | *NPCNA-Prev* | | | | |
| **4d1** | 0.2073 | 10,944,282,367 | -0.4052 | 478,222,298 | 13,390,224,353 |
| **4d2** | 0.2064 | 11,994,192,004 | -0.4573 | **551,627,529** | **15,445,570,812** |
| **8d1** | 0.1901 | 7,703,616,656 | -0.1550 | 292,960,231 | 8,202,886,457 |
| **8d2** | 0.1763 | 7,068,101,620 | -0.0790 | 236,102,494 | 6,610,869,827 |
| **8d4** | **0.1739** | **6,871,884,850** | **-0.0527** | 216,087,916 | 6,050,461,648 |

**Table 4.6: NPCNA and NPCNA-Prev experimental results using different counter configurations.**

# Chapter 5

# Conclusions and Future Work

Building a memory system that can keep pace with faster CPUs has been a computer architectural design challenge for decades. In particular, minimizing the L1 on-chip cache misses will enhance processor performance substantially. However, performance measurements of memory should not only focus on miss latency, but also on memory bandwidth. This thesis investigates selective line replacement strategies and different active data management techniques with the goal of reducing average access times for L1 data caches. Knowing that a small number of instructions are responsible for a large number of misses, the selective line replacement approach effectively saves memory bandwidth by restricting certain data block references from entering the cache. Multi-lateral caches, namely the NTS and PCS caches, utilize reuse information of the EA or PC to allocate data blocks into either temporal or non-temporal caches, and improve miss rates dramatically [22][23][25]. To take advantages of these techniques, I have explored several hybrid cache designs, namely: NTSCNA, NTSPCS, and NPCNA.

Overall, the NTSPCS gives the best speedup over a single 8K direct-mapped cache. However, because NTSPCS and NTS have similar performance gains, the NTSPCS scheme does not seem too useful (since it is more complex). Although initially,

the NTSCNA cache does not perform as well as the base cache, NTS, or the NTSPCS cache, its miss rate can be reduced by lowering the number of CNA instructions marked. NTSCNA is able to maintain the low miss rate of the NTS cache with the 8d1 counter configuration yet save significant amount of memory bandwidth.

Using the default counter configuration, NPCNA caches degrade performance. Like the NTSCNA scheme, through the NPCNA caches can be improved by marking instructions as CNA less frequently. However, comparing NTSCNA to NPCNA and its variation, NPCNA-Prev, the NTSCNA seems to be a much more attractive caching scheme since it not only performs better, but also reduces the complexity of the cache design. By examining the miss ratio, cache speedup, and memory bandwidth savings for the simulations, the NTSCNA caching scheme is found to be the most promising caching scheme.

Even though the NTSCNA 8d1 is a smaller cache, it performs almost as well as a single direct-mapped cache. The size of the direct-mapped cache is nearly twice that of the NTSCNA cache; however, the difference in average miss rate is only 0.005. As a result, the extra space on the die can be used toward other resources such as data forwarding and branch prediction, or eliminate entirely to save energy. Although the NTSCNA scheme requires more hardware support than the NTS, the large amount of savings in memory bandwidth allows architects to develop more aggressive latency tolerance techniques. These techniques can help minimize the large disparity between processor and memory speeds, and they include prefetching, streaming, multithreading, and speculative loads [2]. For advanced architectural designs, this caching scheme can reduce the miss rate and lower the memory bandwidth, thus helping the system to

overcome the memory performance gap. The NTS cache, on the other hand, will be more useful for simpler processor designs in which memory bandwidth is not too scarce.

In this thesis, I only proposed incorporating the CNA algorithm in multi-lateral caches that utilize reuse information. Possible future projects can try to study the effectiveness of integrating the CNA scheme into other multi-lateral caches that do not use previous reuse information, such as the Assist and Victim caches. Also, as mentioned in the last chapter, the cause of the increase in miss rate for the NTSCNA scheme compared to its base model can be explained by the frequent interferences of the CNA algorithm and the reuse characteristics of the EA. One can try to explore the behavior of such interferences and find ways to eliminate these additional misses. In addition, the prediction strategies used in the NTSPCS and NPCNA models are too simple. There might be room for improvement if more advanced prediction algorithms were used. By investigating these algorithms with the NTSPCS and NPCNA models, one can hope to take full advantage of the reuse information of both the EA and PC.

# Bibliography

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau and R. Gupta, “Predictability of Load/Store Instruction Latencies,” in *Proceeding of the 26th Annual International Symposium on Microarchitectures*, Austin, Texas (December 1-3, 1993), pp. 139-152.

[2] D. Agarwal, W. Liu, and D. Yeung. "Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption," in *Proceeding 4th Workshop on Complexity-Effective Design*, June 2003.

[3] Jean-Loup Baer, Tien-Fu Chen. “Effective Hardware-based Data Prefetching for High-performance Processors,” *IEEE Transactions on Computers*, Volume 44, Issue 5, p.609 623, May, 1995.

[4] J. -L. Baer and W.-H. Wang. “Multi-level cache hierarchies: Organizations, Protocols and Performance,” *Journal of Parallel and Distributed computing*, 6(3): 451-476, 1989.

[5] D. Burger, A. Kagi, and J. R. Goodman. “Memory bandwidth limitations of future microprocessors,” *in Proceedings of the 23rd International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[6] Tien-Fu Chen and Jean-Loup Baer. “Reducing Memory Latency via Non-blocking and Prefetching Catches,” in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[7] R. Cucchiara, M. Piccardi, A. Prati. “Neighbor cache prefetching for multimedia image and video processing,” in *Press on IEEE Transactions on Multimedia*, 2004.

[8] C. Ding and K. Kennedy. “Memory Bandwidth Bottleneck and its Amelioration by a Compiler,” *Technical report, Rice University*, May 1999. Submitted for publication.

[9] J.W.C. Fu, J.H. Patel. “Data prefetching in multiprocessor vector cache memories,” in *Proceedings of IEEE/ACM Intl. Symp. on Computer Architecture (ISCA)*, pp. 54–63, May 1991.

[10] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. *Morgan Kaufmann, 3rd edition*, 2003.

[11] Intel Analysis Tools for Object Modifications (Intel ATOM). http://www.intel.com/cd/software/products/asmo-na/eng/enabling/219608.htm.

[12] N.P. Jouppi. "Improving Direct-mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," in *Proceedings of the 17th Annual Intl. Symposium on Computer Architecture*, May 1990.

[13] Todd C. Mowry, Monica S. Lam, Anoop Gupta. "Design and Evaluation of a Compiler Algorithm for Prefetching," *ACM SIGPLAN Notices, Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Volume 27 Issue 9, September 1992.

[14] J. M. Mulder, N. T. Quach and M. J. Flynn. "An Area Model for On-Chip Memories and its Application," IEEE Journal of Solid-State Circuits, vol. 26, no. 2 (February 1991), pp. 98-105.

[15] E. Rashid et al. "A CMOS RISC CPU with On-Chip Parallel Cache," *ISCC Digest of Papers*, February 1994, pp. 210-211.

[16] J. A. Rivers and E. S. Davidson. "Reducing Conflicts in Direct-Mapped Cache with a Temporality-Based Design," in *Proceedings of the 1996 ICPP*, vol. I., Bloomingdale, IL, August 12-16, 1996, pp. 154 – 163.

[17] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. "Utilizing Reuse Information in Data Cache Management," in P*roceedings of the 12th ACM International Conference on Supercomputing,* July, 1998.

[18] SPEC CPU2000. http://www.spec.org/cpu2000.

[19] A. Smith. "Cache Memories," *ACM Computing Surveys (CSUR)*, Volume 14, Issue 3, September 1982.

[20] A. Srivastava and A. Eustace. "ATOM: A system for building customized program analysis tools," in *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196--205. ACM, 1994. 11.

[21] Pieter Struik, Pieter van der Wolf, Andy D. Pimentel. "A Combined Hardware/Software Solution for Stream Prefetching in Multimedia Applications," in *Proceeding of SPIE Multimedia Hardware Architectures*, pages 120-130, Jan. 1998.

[22] Edward S. Tam, Jude A. Rivers, Vijayalakshmi Srinivasan, Gary S. Tyson and Edward S. Davidson, "Active Management of Data Caches by Exploiting Reuse

Information," *IEEE Transactions on Computers*, Vol 48, No 11, pp. 1244-1259, Nov 1999.

[23] E. S. Tam, J. A. Rivers, and E. S. Davidson, "Flexible Timing Simulation of Multiple Cache Configurations," *Technical Report CSE-TR-348-97*, University of Michigan, November 1997.

[24] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "Managing Data Caches using Selective Cache Line Replacement," *Journal of Parallel Programming*, Vol. 25, No. 3, pp. 213--242, June 1997.

[25] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson. "mlcache: A flexible Multi-lateral Cache Simulator," in *Proceedings of MASCOTS'98*, pages 19-26, 1998.

[26] G. Tyson, M. Farrens, J. Matthews and A. Pleszkun, "A Modified Approach to Data Cache Management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, Michigan, United States 1995, pp. 93-103.

[27] T. Wada, S. Rajan and S. A. Przybylski, "An Analytical Access Time Model for On-Chip Cache Memories," *IEEE Journal of Solid-State Circuits*, vol. 27, no.8 (August 1992), pp. 1147-1156.

[28] Joshua J. Yi, Resit Sendag, and David J. Lilja. "The Spatial Characteristics of Load Instructions," *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report,* No. ARCTiC 02-10, October, 2002.

[29] Daniel F. Zucker, Michael J. Flynn, Ruby B. Lee. "A Comparison of Hardware Prefetching Techniques For Multimedia Benchmarks," *1995 International Conference on Multimedia Computing and Systems*.

# Appendix A

# Mlcache Configurations File

```
8192    #A_cache_size_in_bytes
32      #A_cache_blocksize_in_bytes
32      #A_cache_subblocksize_in_bytes
1       #A_cache_associativity_(number_of_sets)
l       #A_cache_replacement_policy:l=LRU,f=FIFO,r=RANDOM
d       #A_cache_fetch_policy:d=DEMAND,for_others_see_fetch.c

1024    #B_cache_size_in_bytes
32      #B_cache_blocksize_in_bytes
32      #B_cache_subblocksize_in_bytes
32      #B_cache_associativity_(number_of_sets)
l       #B_cache_replacement_policy:l=LRU,f=FIFO,r=RANDOM
d       #B_cache_fetch_policy:d=DEMAND,for_others_see_fetch.c

4       #word_size
18      #main_memory_read_latency
18      #main_memory_write_latency
1       #cache_hit_latency
0       #move_time_for_blocks_moving_between_A_and_B

1       #use_r/w_ports:1_for_combined_r/w_ports,0_for_separate
4       #number_of_r/w_cache_ports
0       #number_of_read_cache_ports
0       #number_of_write_cache_ports
1       #return_policy:1_for_requested_word_first,0_for_first_word_first

8       #CPU-to-cache_bus_width_in_bytes
32      #cache-to-memory_bus_width_in_bytes

100     #NOA;set_high_for_nonblocking

0       #TIMING_MODEL;0=Full_LE,1=LE-Nominal,2=NO_timing

32      #extra_parameters,e.g._DU_size

256     #size_of_PCtable;0=UNLIMITPC;eg.256_entries

4       #cna_counter_size_of_PC,4_is_default
1       #PC_counter_decrease_count_for_cache_hit;-1=RESET
```

```
4	#cna_pcs_counter_size,4_is_default
1	#Number_of_decrease_for_each_nts_used;-1=reset_to_zero
1	#Number_of_increase_for_each_pcs_used;-1=reset_to_counter_size
```