1-1-2005

Improving Instruction Fetch Rate with Code Pattern Cache for Superscalar Architecture

Azam Muhammad Beg

IMPROVING INSTRUCTION FETCH RATE WITH CODE PATTERN CACHE

FOR SUPERSCALAR ARCHITECTURE

By

Azam Muhammad Beg

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

August 2005

IMPROVING INSTRUCTION FETCH RATE WITH CODE PATTERN CACHE

FOR SUPERSCALAR ARCHITECTURE

By

Azam Muhammad Beg

Approved:

Yul Chu Assistant Professor of Electrical and Computer Engineering (Director of Dissertation)	Justin Davis Assistant Professor of Electrical and Computer Engineering (Committee Member)
Nicolas H. Younan Professor of Electrical and Computer Engineering (Committee Member/Graduate Program Director)	Susan M. Bridges Professor of Computer Science and Engineering (Committee Member)

Kirk H. Schulz
Dean of the Bagley College of Engineering

Name: Azam Muhammad Beg

Date of Degree: August 6, 2005

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. Yul Chu

Title of Study: IMPROVING INSTRUCTION FETCH RATE WITH CODE-PATTERN CACHE FOR SUPERSCALAR ARCHITECTURE.

Pages in Study: 125

Candidate for Degree of Philosophy

In the past, instruction fetch speeds have been improved by using cache schemes that capture the actual program flow. In this dissertation, we present the architecture of a new instruction cache named *code pattern cache* (CPC); the cache is used with superscalar processors.

CPC's operation is based on the fundamental principles that: common programs tend to repeat their execution patterns; and efficient storage of a program flow can enhance the performance of an instruction fetch mechanism. CPC saves basic blocks (sets of instructions separated by control instructions) and their boundary addresses while the code is running. Basic blocks and their addresses are stored in two separate

structures, called *block pointer cache* (BPC) and *basic block cache* (BBC), respectively. Later, if the same basic block sequence is expected to execute, it is fetched from CPC, instead of the instruction cache; this mechanism results in higher likelihood of delivering a larger number of instructions in every clock cycle.

We developed single and multi-threaded simulators for TC, BC, and CPC, and used them with 10 SPECint2000 benchmarks. The simulation results demonstrated CPC's advantage over TC and BC, in terms of *trace miss rate* and *average trace length*. Additionally, we used cache models to quantify the timing, area, and power for the three cache schemes. Using an aggregate performance index that combined the simulation and modeling results, CPC was shown to perform better than both TC and BC.

During our research, each of the TC-, BC-, or CPC-configurations took 4-6 hours to simulate, so performance comparison of these caches proved to be a very time-consuming process. Neural network models (NNM's) can be time-efficient alternatives to simulations, so we studied their feasibility to represent the cache behavior. We developed two NNM's, one to predict the trace miss rate and the other to predict the average trace length for the three caches. The NNM's modeled the caches with reasonable accuracy, and produced results in a fraction of a second.

DEDICATION

I dedicate this research to my loving parents and wife.

ACKNOWLEDGMENTS

First and foremost, I thank Allah, the God, the Most Merciful, the Beneficent for blessing me with the countless bounties of abilities and resources in this life.

I feel highly indebted to my advisor Dr. Yul Chu for providing me with constant guidance and supervision and for pushing me, at times to accomplish more than what I would have otherwise settled for. I thank my committee members Drs. Younan, Davis, and Bridges for their insightful commentary on my research. I am thankful to the ECE department for the much needed financial support in the last phase of my research.

I owe my deepest gratitude to my parents for the values they instilled in me while bringing me up and for teaching me the lessons about hard work and persistence.

I would also like to thank my wife Shabana; without her support and patience, it would have not been possible to complete this arduous journey. I appreciate my whole family including my children Raahim and Rahma for letting me busy with my studies during many, many evenings

and nights in the last 8 years, especially during the last few months of my PhD program.

I also value the “Just Do It” words of encouragement, I often received from my friends Drs. Amr and Ashraf.

Finally, I would also like to say a word of appreciation to everyone who helped me with my PhD degree in one way or other.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

Stated simply, the steps in a program execution are: (1) fetching an instruction; (2) reading the related data; (3) performing calculations; (4) and storing the results (to memory or register file, if needed); then going back to the 1st step to fetch the next instruction [Hennessy & Patterson 2003]. In order to perform these tasks, a modern processor can be organized as a structure that is divided into an instruction fetch unit (called producer) and an instruction execution unit (called consumer). The producer and the consumer are separated by instruction issue buffers (collectively called the instruction window). The goal is that the producer issues the instructions at the highest possible rate while the consumer attempts to execute the instructions as fast as it can [Hennessy & Patterson 2003]. With the ongoing validity of nearly 40-year old Moore's Law [Moore 1965], an ever-increasing number of devices are available to the processor designers, who are faced with the constant challenge of balancing the performance, design complexity, testability, manufacturability, and all the related costs.

In this dissertation, we present a new method of improving *producer* performance by introducing a new instruction cache, called *code pattern cache* (CPC). CPC's functionality is based on the execution patterns (dynamic nature) of programs. In plain terms, CPC's salient features are:

- It exploits the empirical observation that a program's sets of instructions (blocks) come in varying sizes
- It does not require that the same set of instructions be stored in multiple locations
- It does not need to replicate the cache structures
- It makes use of the (traditional) principle of cache associativity
- It increases the likelihood of finding previously executed sequences of instructions (*traces*) inside the cache, and every time it finds a sequence, CPC-traces tend to contain more instructions than existing schemes
- It maintains its ability to deliver traces even while serving multiple *consumers*

The other contribution of this dissertation is the creation of neural network models (NNM's) for CPC and similar cache schemes. The models can be used to perform what-if analyses of cache design space without running the time-consuming simulations.

CPC's performance measures are:

- The likelihood of finding instructions (*hits*) in cache
- The number of instructions found with each hit
- The die area it takes to implement the cache
- The power consumed by the cache
- The time it takes to access the cache

This chapter first broadly covers the aspects of microprocessor performance and different approaches for its improvement, and then discusses the motivation and contributions of this dissertation in detail.

1.1 Background

1.1.1 Microprocessor Performance

In order to speed up program execution, one may simply use larger or faster semiconductor circuits that make use of newer and smaller transistors. One use of increased availability of transistors is to increase the number of instruction execution units. The number of execution units determines the maximum number of instructions that can be issued by the *producer* in one processor clock cycle [Hennessy & Patterson 2003]. A processor with multiple execution units is called a *superscalar* processor. Using a larger number of execution units results in increased die size and power consumption, which may not be desirable for cost-effective designs. Besides utilizing faster circuits,

today's high-performance processors use many techniques, such as *caching* and *branch prediction*; these two techniques make use of the pragmatic behavior of the programs, which assume that their execution behavior is not random, and that it follows certain patterns. The branch prediction helps fetch instructions from memory (and sometimes even execute them) in advance without knowing the outcome of the current instruction [Hennessy & Patterson 2003]. In other words, branch prediction exploits the regularity in the program flow. Caching operation is based on the observation that the programs tend to access contiguous locations in memory (*spatial locality*) or the recently accessed memory locations repetitively (*temporal locality*) [Hennessy & Patterson 2003]. This program behavior results in low *latency* (how fast the memory contents are available) and higher *bandwidth* (how much data is readily available) for caches. Effectively, the caches try to approximate the availability of ideally large memory that the programmers expect [Hennessy & Patterson 2003]. Another reason why the memory latency is critical is that the processor speed has risen at a rate much higher than memory speed. This increasing gap is a constant challenge for processor system designers. The issue is particularly significant for applications that require large memory bandwidth, such as digital image-processing, especially if the data needs to be transferred over a simple, standard

interface. So the issue of memory latency becomes ever more important [McBader & Lee 2003].

One or more levels of caches can be used between the processor and the main memory; placing fast caches close to the processor reduces memory latency by storing frequently or recently accessed data and instructions. The caches closest to the processor are fast but small in size, whereas longer latency caches are larger in size and store less frequently accessed data and instructions [Shanley & Anderson 1995].

The caches go only so far with the alleviation of memory latency constraints because of cache misses and the resulting processor stalls. So it becomes imperative to manage the caches efficiently. Two important aspects of cache management are: (1) when and how much data to bring into cache (*pre-fetching*); and (2) what to retain in cache and what to replace [Hennessy & Patterson 2003].

1.1.2 Overview of Thread-Level Parallelism

Kavi et al. (1995) define a *thread* as a set of instructions that starts execution at its first instruction and continues execution without interruption. A single program can be executed on multiple processors that have shared code and (most of the) address space. Sharing of code and data in this manner is traditionally called *threading*. These days, threading also refers to execution in multiple locations even when the

address space is not shared [Hennessy & Patterson 2003]. A programmer can identify independent threads or he can use a compiler for this purpose. Threads can be large, fully independent programs or parts of a single program (for example, parallel iterations in a loop). *Parallelism* is defined as the potential of simultaneous execution and *thread-level parallelism* (TLP) is "logically structured as separate threads of execution." The exploitation of thread-level parallelism is an effective way of overcoming the limitations of memory latency [Hennessy & Patterson 2003]. *Instruction level parallelism* (ILP), in contrast, exploits the ability to issue multiple instructions in a cycle. In hardware-related implementation, opportunities for ILP are identified and scheduled by hardware; whereas, software-centric ILP depends on static scheduling by a (*very long instruction word*) compiler. ILP's main advantage is that it makes use of parallelism without requiring re-writing of the existing programs [Schlansker et al. 1997].

Multi-threading (MT) is a technique that allows multiple threads to share the execution units of a single processor in a parallel fashion. The hardware must support the switching of threads efficiently. To enable MT, some components of a processor (for example, the register file and the program counter) need to be replicated. Sharing of memory can be done via the *virtual memory* technique [Hennessy & Patterson 2003].

(Virtual memory (VM) automates the job of moving program and data between the main memory and secondary storage. One of VM's advantages is the ease for a programmer, especially, when his program code and data sizes exceed the physically available memory [Jacob & Mudge 1998]).

Fine-grain MT and *coarse-grain MT* are two main approaches to MT. Fine-grain MT allows switching of threads on every instruction, in a round-robin fashion (while skipping any stalled threads). On the other hand, coarse-grain MT switches threads when the currently executing thread stalls for many cycles due to for example, a *miss* on the cache closest to the main memory [Hennessy & Patterson 2003].

Figure 1 shows the states of 4 execution units (EU1-EU4) of a (single-threaded) superscalar processor in several cycles. Different executions units are used every cycle for the same thread T1. A *used* execution unit is represented by a box containing letter T followed by the thread number; an *unused* execution is shown as an empty box (Figure 2, Figure 3, and Figure 4 also follow the same conventions). In Figure 1, we see that in cycle *n,* EU1 and EU2 are used, while EU3 and EU4 remain unused; in cycle *n+1,* EU2 and EU3 are used, while EU1 and EU4 remain unused; and so on. Due to a stall on T1 during cycle *n+4*, all four execution units remain unused.

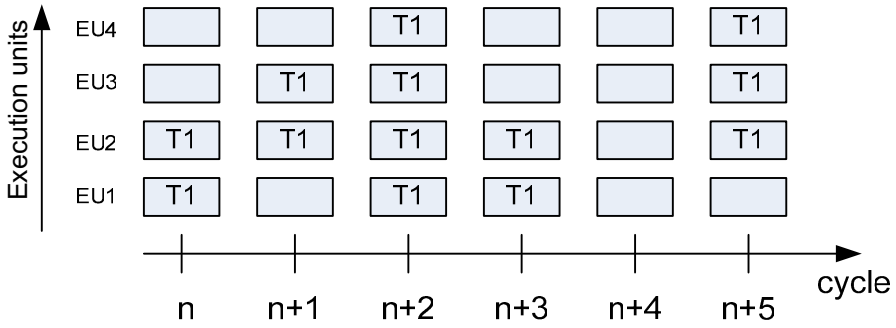

Figure 1. States of 4 execution units of a superscalar processor executing a single thread (T1)

The states of execution units in a fine-grain MT processor are shown in Figure 2.

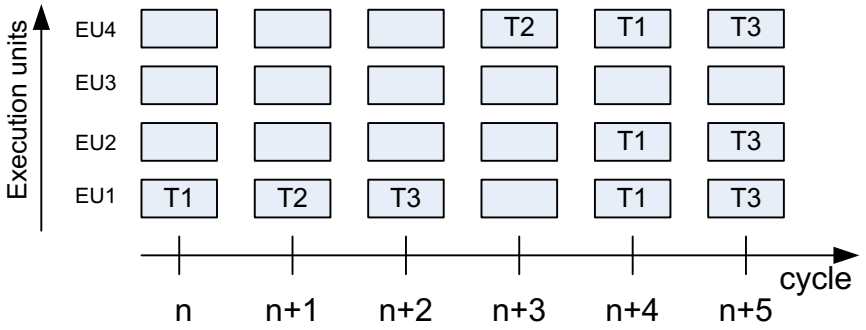

Figure 2. States of 4 execution units of a fine-grain MT processor executing 3 threads (T1, T2, T3); threads switch in round-robin fashion every cycle.

Three threads T1, T2, T3 get executed on the processor. In cycle *n,* two executions EU1 and EU2 are used by thread T1; in the next cycle *n+1*, T2 uses the same execution units; and so on. This thread-switching continues in a round-robin fashion, unless some thread is

skipped due to a stall; this happens in cycle *n+5,* where T2 is passed over by T3.

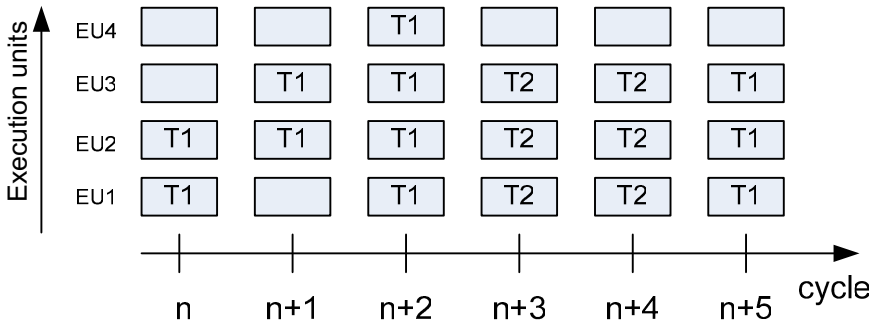

Figure 3. States of 4 execution units of coarse-grain MT processor executing two threads (T1 and T2); thread switches from T1 to T2 in cycle n+3 due to long stall on thread T1.

The coarse-grained MT does not switch from the currently running thread to the next unless there is a long (multi-cycle) stall on the current thread. For example, in Figure 3, thread T1 keeps occupying different execution units during *n, n+1,* and *n+2* cycles, until a stall on T1 causes T2 to start executing in cycle *n+3*.

In the case of simultaneous multi-threading (SMT), TLP and ILP techniques are combined concurrently [Hennessy & Patterson 2003]. SMT allows multiple issues of independent threads to multiple execution units per cycle [Tullsen et al. 1995]. Processor resources in an SMT processor are shared among threads on per-cycle basis. But, as the processor has to hold instructions from multiple threads, larger issue

instruction queues may be required [El-Moursy & Albonesi 2003]. (Instructions are held in an instruction queue before being sent to execution units). The term Hyper-Threading is used for the implementation of dual-thread SMT on Intel's Pentium-4 and Xeon processors [Intel 1997], [Intel 2001], [Marr 2002]. The SMT processor of Figure 4 shows that during a given cycle, more than one thread is allowed to execute. For example, T1 and T2 execute simultaneously in cycle *n*, T1 and T3 execute in cycle *n+1*, and so on. The result can be better utilization of execution ('consumer') resources.

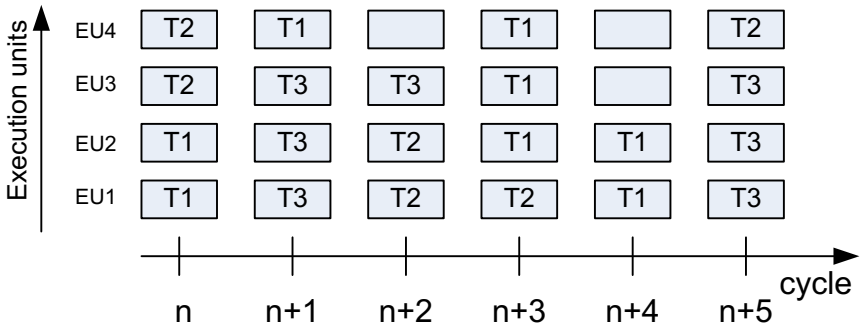

Figure 4. States of 4 execution units of a simultaneous multi-threading (SMT) processor. Based on the availability of execution units, instructions from one or more threads are allowed to execute every cycle.

An MT processor alters the way the memory is accessed. Cache effectiveness is reduced because of the changed locality of reference [Lioupis & Milios 1997]. To address this issue, an MT architecture presented by Govindarajan et al. (1995) had separate instruction and

data caches. Lioupis & Milios (1997) studied behavior of a single-thread in an MT processor with different cache configurations. They proposed a pipelined interface between the cache and the rest of the memory hierarchy for better cache performance.

1.1.3 Basic Blocks

A *basic block* is a set of contiguous instructions that contains only a single control instruction such as a conditional or an unconditional jump, a return, or a call. The control instruction is the last instruction of a basic block, and is also called the *block tail*. The beginning of a basic block is called its *block head*. Block head is also the destination of a control transfer instruction [Ozturk et al. 2005]. In this dissertation, basic blocks have no size limitations other than the cache capacity.

Address	Instruction	Comments
0000	ADD	Block *0* head on the 1st instruction of the program
0008	ADD	
0010	**BNE**	Block *0* tail due to a conditional jump
	:	
	:	
0030	ADD	Block *m* head
0038	SUB	
0040	ADD	
0048	DIV	
0050	**J**	Block *m* tail due to an unconditional jump
	:	
	:	
0058	ADD	Block *n* head
0060	MULT	
0068	OR	
0070	ADD	
0078	**BEQ**	Block *n* tail due to a conditional jump
1000	SUB	Block *x* head
1008	**HLT**	Block *x* tail; halt instruction terminates the program

Figure 5. Four basic blocks from a sample program are shown. Block beginnings (heads) and (tails) are also identified. (Addresses are shown in hexadecimal format).

A sample assembly program with some control instructions is shown in Figure 5. The program contains conditional and unconditional jumps. The beginning and the end of each block is also indicated. The code in this example shows four basic blocks. Basic block 0 starts with the 1st instruction of the program and ends with a conditional branch (BNE). The *head* address of this block is 0000 and the *tail* address is 0010. The length of this block is 3 instructions. The *head* of block *m* at address 0030 is the destination of a conditional or unconditional jump from (the same or a different) basic block. The *tail* of this block at address

0050 is determined by an unconditional jump instruction (J). The length of this block is 5 instructions. Similarly, block *n* has its *head* and *tail* at addresses 0058 and 0078, respectively. The last block *x* starts at address 1000; the block ends with a halt (HLT) instruction at address 1008.

1.1.4 Conventional Instruction Cache

The basic data unit of conventional instruction cache (IC) is a *cache line* that stores a set of memory-adjacent instructions. The usual cache line lengths are 16 to 64 bytes. IC, although simple to implement, tends to exhibit high latency and low bandwidth. Typically single-ported reads limit IC bandwidth to a single basic block because of a jump to a non-adjacent memory location. This type of jump is called a *taken-branch*. A simplified block diagram of a superscalar processor with an IC is shown in Figure 6 [Hennessy & Patterson 2003]. The instructions are provided from the IC to the decoder. Only one cache line can be delivered per cycle. The basic blocks beyond a taken-branch are fetched in the following cycle as illustrated by block #5 in cycle 2 in Figure 7. (In Figure 7, Figure 9, and Figure 11, a number before an 'x' represents the instruction count in a basic block. The upper case 'A' or 'B' used as a suffix to a block number indicates that the block is split over two cache lines. The arrows indicate the instruction flow.)

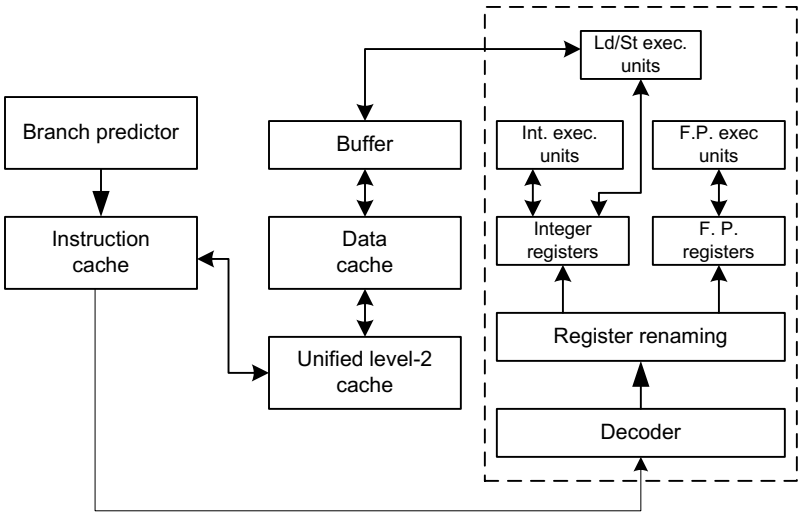

Figure 6: A superscalar processor with an instruction cache (IC)

Assuming that no pre-fetch buffer is present, the example in Figure 7 requires 5 cycles to fetch the two contiguous blocks (#0, #1) and three non-contiguous blocks (#5, #6, #10). Note that block #10 straddles across 2 cache lines. So the fetching of this block is split over two cycles.

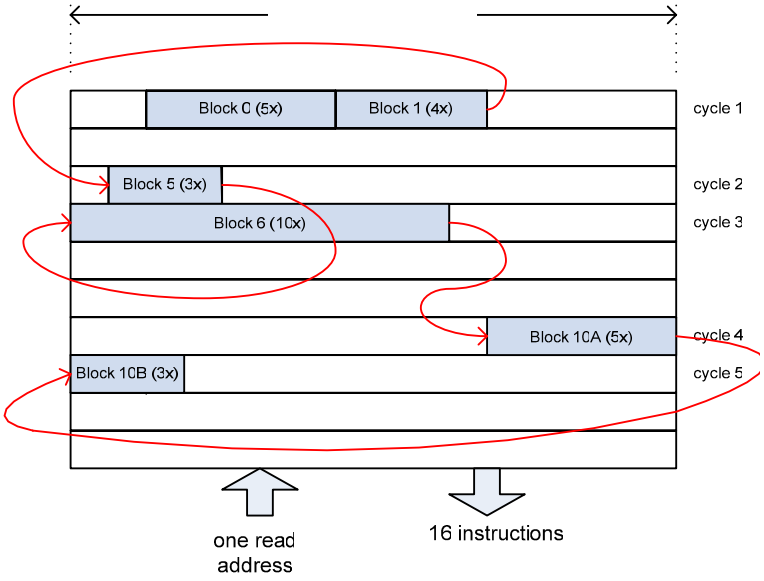

Figure 7. Basic blocks in the lines of an IC: 5 cycles are required to fetch the non-contiguous basic blocks due to taken branches. (The arrows show the sequence of instruction execution).

Much research has been done on techniques that improve bandwidth beyond IC. For example, Dutta & Franklin (1995) (1999) used a tree-like subgraph for an executed program to predict multiple branches in a single prediction. Hao, et al's (1996) block-based architecture depended on compile-time and hardware-based solutions. However, they introduced redundancy in storage when they combined basic blocks to create larger blocks.

As mentioned earlier, branches can make a program jump out of a cache line. When *taken,* the branches in the middle of a cache line leave many unexecuted instructions and hence cost additional read cycles to

fetch another line. (A taken-branch usually jumps to a non-contiguous program location). Keeping this IC behavior in view, techniques for improved instruction fetching have been presented by Conte et al. (1995), Hily & Seznec (1996), McFarling (1993), Wallace & Bagherzadeh (1998), and Yeh & Patt (1992). Conte et al's (1995) *collapsing buffer* scheme was able to align non-adjacent basic blocks up to 90% of the time. McFarling (1993) combined different branch predictors in such a way that only the most accurate prediction was used for a branch. Hily & Seznec's (1996) study on three common branch predictors included use of branch history tables whose sizes proportionally increased with the number of programs executing in parallel. Wallace & Bagherzadeh's (1998) instruction fetch mechanism involved a *dual branch target buffer* that tried to predict the starting addresses of the next two cache lines. Yeh & Patt's (1992) *adaptive branch prediction* scheme used two levels of branch prediction by looking at the n-level history of the last few branches. All of these techniques still limited the instructions fetched per cycle to one or two basic blocks. For better performance, more basic blocks need to be fetched every cycle which is possible in Rotenberg et al's (1999) *trace cache* (TC) and other follow-up schemes [Black et al. 1999], [Jourdan et al. 2000].

1.2 Related Works

1.2.1 Trace Cache

Rotenberg, *et al*'s (1999) TC bypassed IC's fundamental instruction limit due to taken branches and resulted in increased bandwidth and reduced latency of instruction decoding. TC also addressed some of the issues present in the previous schemes (refer to previous section) [Conte et al. 1995]. TC captured instructions as they were executed. The matching of the starting address of a TC line and the predictions for branches inside the line are the two conditions that cause the delivery of instructions (to the instruction decoder) from TC, instead of IC. TC made it possible to fetch multiple basic blocks in one cycle (Refer to Figure 9) [Black et al. 1999], [Gummaraju & Franklin 2000], [Howard & Lipasti 1999], [Jourdan et al. 2000], [Patel et al. 1998], [Patel et al. 1999].

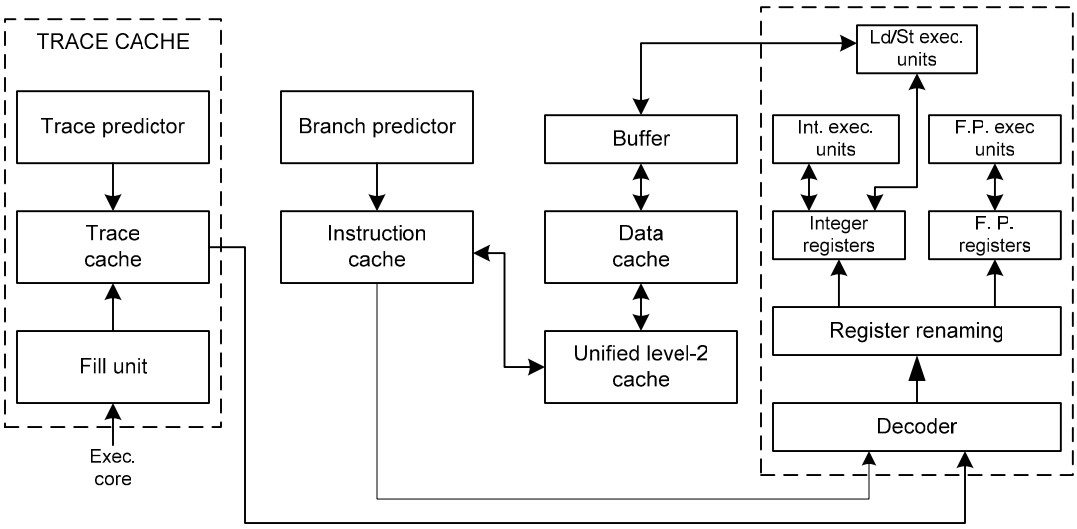

Figure 8. A superscalar processor with a trace cache (TC)

The block diagram of a TC-based superscalar microarchitecture is shown in Figure 8. A single TC-trace may contain more than one block. When there is a TC-hit, more than one basic block can be delivered to the decoder in the same cycle. In case of a miss on TC, a cache line is brought from the IC.

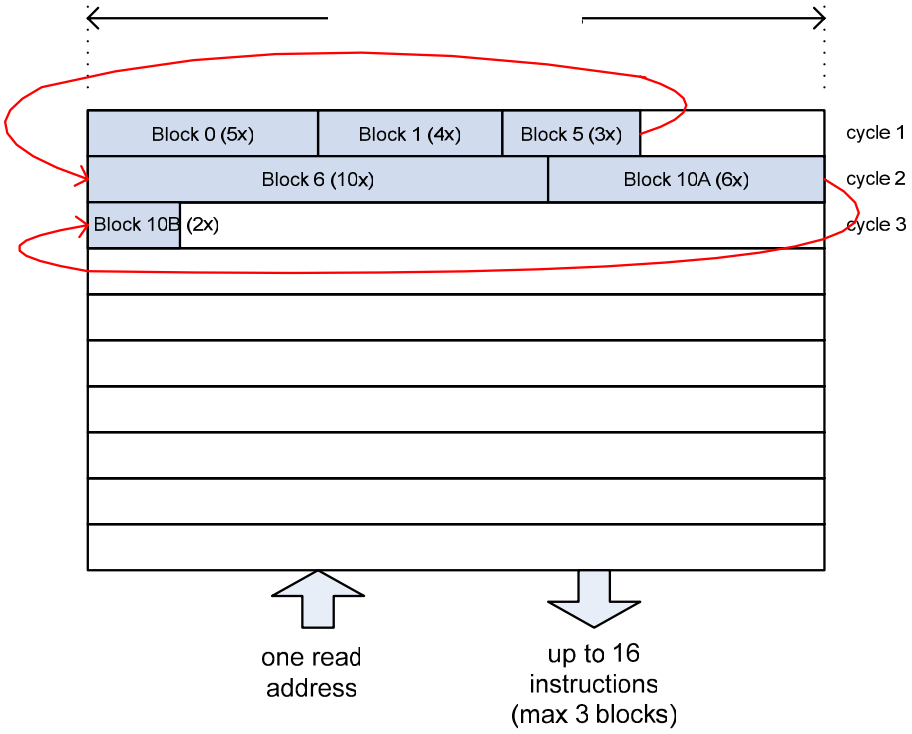

Figure 9. Basic blocks in the lines of a trace cache (TC): A maximum of 16 instructions or 3 basic blocks are stored in a TC line. 'Instruction holes' are left at the end of the first cache line. In cycle-1, 3 basic blocks are delivered. (The arrows show the sequence of instruction execution).

In the TC in Figure 9, a cache line contains a maximum of 3 basic blocks or 16 instructions. Blocks #0, #1, and #5 are all fetched in one cycle. Three blocks in the first line have only 12 instructions, so space for three instructions is left unused. In other words, three 'instruction holes' are left at the end of the cache line. Although block #10 still happens to cross the cache line boundary, fetching of up to 16 instructions is now possible in cycle 2. In case of TC, only 3 (instead of 5)

cycles are needed to fetch the same set of instructions as the conventional IC (of Figure 7).

1.2.2 Block Caches

A variation of TC was Black, et al's (1999) cache scheme called *block cache* (BC). The scheme included identification of individual blocks in the stored traces. The block identifiers (pointers) were used to assemble the traces, on a trace hit. In BC, two separate cache structures were used, one (called *block cache)* to store the basic blocks and the other (called *trace table*) to store the block pointers. The blocks were replicated 4 times in Black et al's (1999) scheme. Each line in the block cache stored a single basic block. The assumption that basic blocks were all the same width caused an increase in the likelihood of block fragmentation. (Refer to Figure 11 for the examples of fragmented blocks). Black et al. (1999) reported that, with perfect branch prediction, BC helped a processor complete 7% more instructions per cycle than TC. However, they did not compare BC's trace miss rate and average trace length with TC.

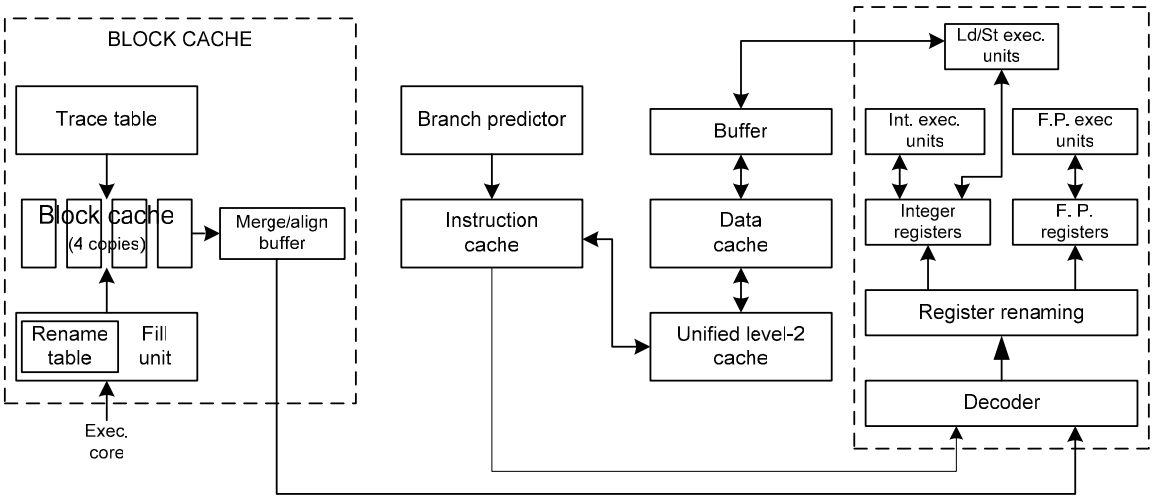

Figure 10. A superscalar processor with a block cache (BC)

The block diagram in Figure 10 shows a superscalar processor connected to BC. The trace table is used to determine a BC-hit or a miss. On a hit, the basic blocks are fetched from the block cache. A complete trace is built by passing instructions through the merge/align buffer being sent to the decoder. A BC-miss causes delivery of instructions from the IC.

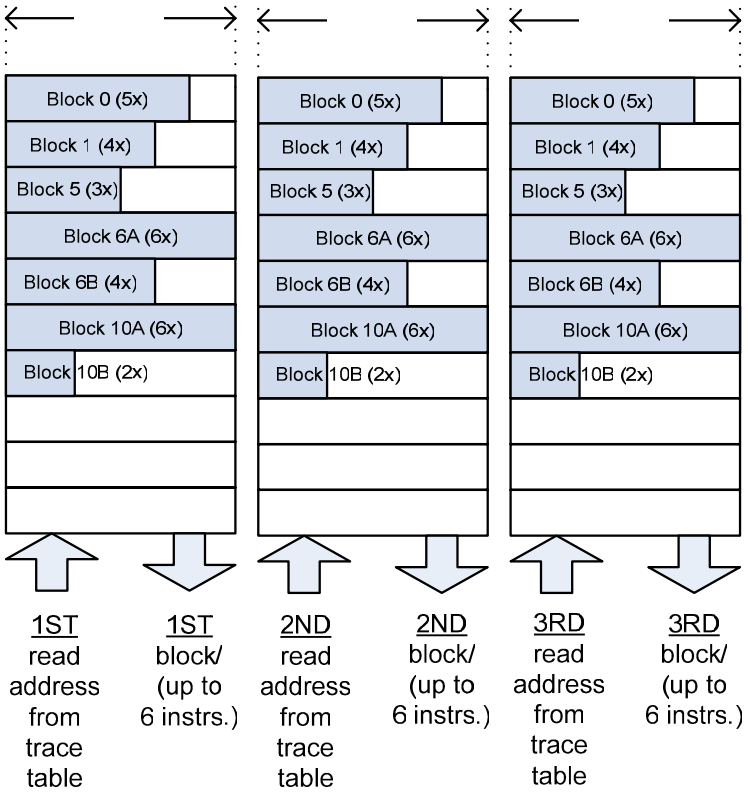

Figure 11. Basic blocks in block cache (BC): 3 blocks are fetched per cycle; each block is up to 6 instructions long. As compared to TC, there is a potential increase in block fragmentation, as well as in more 'instruction holes' being left in block cache lines. (Block execution sequence is the same as the examples of Figure 7 and Figure 9, but has been omitted here for clarity).

In Figure 11, three blocks are fetched every cycle with each block containing a maximum of 6 instructions. All blocks that are longer than 6 instructions have to be split over more than one block cache line. When blocks are not multiples of 6 instructions, instruction holes are encountered at the end of block cache lines. In Figure 11, there are two fragmented blocks (#6 and #10) (vs. the TC of Figure 9 that has only one

such block, i.e., block #10). Also, one can observe the redundancy of storage; each basic block must be stored in 3 identical cache structures.

A scheme similar to BC was proposed in a Jourdan, et al. (2000). Their scheme, called *extended block* (XB) cache, stored the instructions (uops) in reverse sequence, giving them the ability to extend any existing XB's. They reported reduced block fragmentation. Black et al. (1999) and Jourdan et al. (2000) reported cache performance results only for single-threaded environments. The XB bandwidth was similar to TC. Due to this marginal improvement over TC and due to XB's significantly complex implementation logic, we will limit our discussion of XB to this section only and will not use XB for performance comparison in our research. Unlike TC, no follow-up research has been reported on either BC or XB, since their introduction.

1.2.3 Modeling Techniques

Simulation models provide a faster method of studying the design or operation of a system compared to actual implementation [Smith 1994]. Usually, *mathematical* or *analytical models* comprise a set of mathematical equations. NNM's, on the other hand, are made up of a set of weights that are applied to the model inputs to calculate the outputs. (Chapter IV discusses NNM's in detail). Mathematical models based on response curves (polynomial, spline, etc.) are ineffective with highly non-

linear systems, while NNM's excel with large number of parameters [Stegmayer & Chiotti 2004]. NNM's are robust and provide a good alternative to lookup methods that require storage of all data points in a given data space [Simpson et al. 1997].

In the past, mathematical models and NNM's have been used to model electronic systems and sub-systems. A few examples of mathematical or analytical models are: the model of a program behavior to predict the miss ratio of a fully-associative cache [Singh et al. 1992]; the model of a superscalar processor that included interaction of parallelisms in programs and machines as a performance measure [Noonburg & Shen's 1994); the model for instruction-fetch performance of superscalar processors [Wallace & Bagherzadeh 1998); the miss-ratio model for set associative caches [Harper, et al. 1999); and the model for TC [Hossain et al. 2002). Examples of NNM's are: the NN-based controller to adjust the memory resources in a multi-programming system [Bigus 1994]; the model for analog component behavior [Sobecks et al. 1998]; and the analysis tool that finds bottlenecks in a computer system, such as memory, network, processor, etc. [Gruen & Kubota 2002].

Only in recent years has some research been published dealing with NNM's application to the field of computer architecture. One such example is Jimenez & Lin's (2001) NN-based branch predictor; it does not

suffer from the drawbacks of a conventional branch predictor whose hardware requirements rise exponentially when the branch histories are lengthened. The other example is an NNM for a cache replacement scheme presented by Khalid (1996), and Khalid & Obaidat (2000). The authors used an NNM for predicting the pattern of memory references made by the processor.

The effectiveness with which the NNM's usually model the non-linear and multi-variate systems and the ease of NNM creation are the primary reasons for their use for cache modeling in this dissertation.

CACTI is an analytical model for estimating the area, power, and timing for caches [Wilton & Jouppi 1996]. Since its introduction, CACTI has been used as an estimation tool by several researchers. For example, Batson & Vijaykumar (2001) used this tool to estimate the hit-time for reactive-associative cache; the cache scheme implemented flexible associativity by placing most blocks in direct-mapped positions and reactively displacing only conflicting blocks to set-associative positions. Banakar et al. (2002) used CACTI to compute area and energy for their proposed scratch pad memory, an alternative to cache. Sangireddy et al. (2004) used CACTI to study a low-power technique for cache-based reconfigurable architecture. In our research, we use CACTI to compare

the area, power, and timing requirements for cache structures in TC, BC, and CPC.

1.3 Motivation

Below, we have identified several issues with the current trace-based schemes, namely, TC and BC:

- There is a tendency for TC to have the same set of instructions (full or partial basic blocks) appear in multiple traces. A few examples of percentages of instructions overlapping among traces (for the SPEC2000int [Spec 2000] benchmark programs) are[1]: *crafty* = 25.1%, *mcf* = 38.5%, *bzip* = 79.5%. Redundancy of traces between IC and TC was addressed by Ramirez et al's (2000) scheme, but redundancy of instructions stored in the TC itself was not considered. If the storage redundancy is removed, the cache could be used to store more traces, and hence the miss rate could be reduced.
- TC uses only the beginning address of a trace for matching. Blocks other than the beginning block are not identifiable, so even if the required instructions are present in the trace, the trace is declared a 'miss' and a new trace build is initiated. This rebuilding requires

[1] The traces were extracted from a TC built by modifying *sim-cache* (of SimpleScalar 3.0 tool suite) [Burger 1997], [Burger & Austin 1997]. TC size was fixed at 64 traces while each trace had a maximum of 16 instructions or 4 basic blocks. The benchmark programs [Spec 2000] were run for 200 million instructions.

unnecessary switching from *trace utilization* mode to *build mode.* (In trace utilization mode, the instructions are delivered from TC; and in the build mode, the instructions fetched from the IC are executed and the traces are built/stored in TC). If this mode-switching is reduced, the average number of instructions fetched per cycle can be improved.

- TC traces can leave unused spaces at the end of cache lines if a program (or part of a program) is made up of smaller basic blocks. Cache space can be more efficiently used if the stored traces are of variable, rather than fixed lengths (as in BC), and if the trace lengths can exceed the usual limit of a cache line width.

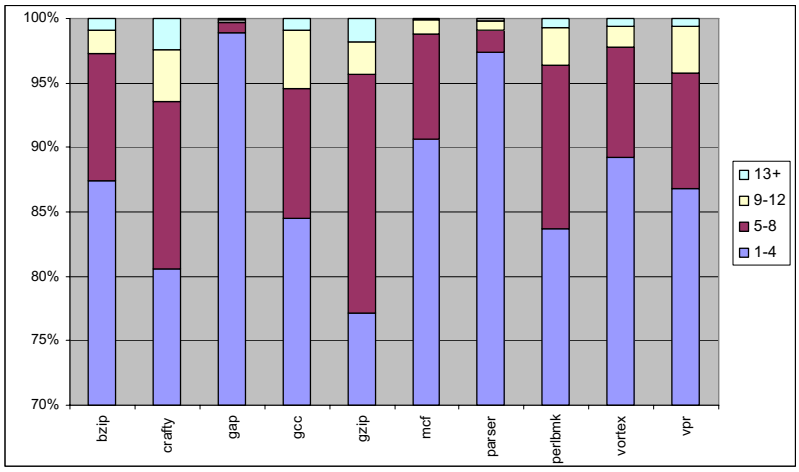

Figure 12. Block length distribution in different SPECint2000 benchmarks

- BC scheme suffers block fragmentation because the basic blocks are assumed to be of fixed length. With the blocks sizes (4-6 instructions per block) assumed in TC/BC designs, some instruction fetch capability may remain under-utilized because, as we can see in Figure 12, up to 23% of the blocks contain more than 4 instructions, and up to 11% of the blocks contain more than 6 instructions. For example, *bzip* has nearly 13% of the blocks longer than 4 instructions, and *crafty* has 19% such blocks. Figure 12 shows the percentage values of block sizes and counts for different SPECint2000 benchmark runs.
- BC potentially has more block-level fragmentation with smaller block sizes (of 4 or so instructions). The fragmentation also happens when a *logical* basic block that is wider than the block cache width is split into more than one *physical* block [Black et al. 1999]. (Refer to the example of Figure 11).
- BC requires storage of same basic blocks in multiple places. Black, et al's (1999) research has BC replicating the blocks 4 times. Redundancy of cache structures, if removed, could reduce the die area and the consumed power.

1.4 Contributions

In this research, we focus on the implementation of new instruction cache architecture that improves the fetch rates beyond what existing trace-based caches have reportedly attained. The following are the salient contributions of this dissertation:

(1) We introduce a new instruction cache scheme called *code pattern cache* (CPC); some of the current trace-based instruction issues that CPC addresses are: (a) eliminating TC's redundancy of instruction storage; (b) removing BC's duplication of caches (thus reducing consumed power and die area); (c) including way-associativity instead of BC's single-way structures; (d) resolving the issue of BC's block fragmentation; (e) allowing the traces to be of variable length; and (f) improving access time (over BC) by enabling simultaneous access to basic blocks and their pointers.

(2) We have developed functional simulators for existing trace-based schemes (TC and BC) and CPC, operating in single-threaded mode. The simulators were developed in VHDL and provided a means to compare the performance of different caches.

(3) We have enabled multi-threaded operation on the VHDL-based simulators for TC, BC, and CPC. The simulators allow instantiation of any number of threads; the only limitation may be the ability of

a simulation platform (VHDL simulator) to complete simulations in a reasonable amount of time.

(4) We have studied the implementation aspects of TC, BC, and CPC, such as power, area, and access time. Comparisons were made for different cache capacities.

(5) We propose an aggregate performance index that combines the simulation results (trace miss rate, average trace length) and modeling results (power, area, access time). The index provides the means to compare the overall performance of TC, BC, and CPC.

(6) We have developed two NNM's for modeling caches, one for predicting their miss rates, and the other for predicting average trace lengths. Each NNM collectively models the behavior of TC, BC, and CPC. The NNM's provide a method that is several orders-of-magnitude faster than simulation for exploring the design space of the three caches. (Until the time of this writing, no other such models for any cache scheme have been reported in the research publications).

1.5 Performance Evaluation

In order to compare TC and BC with CPC, we use several performance metrics. Two of the metrics are: *trace miss rate* and *average trace length*. These metrics are considered to be among the most

appropriate in the context of trace-based caches (TC, BC, etc.). Trace miss rate is the percentage of references when a requested trace was not found in the cache. A smaller value of trace miss rate represents a lower average latency for cache data fetching. Average trace length can be considered to be a measure of how efficiently the cache storage space is being utilized. The longer the traces, the larger the number of instructions fetchable per cycle.

We use only one level of cache in our functional simulators since the main focus of our research is the cache's own performance rather than that of a complete processor system. Simulating only the cache functionality also means that no direct method of calculating the processor-related *instructions completed per cycle* metric is available.

We study the trace miss rate and average trace length by using the instruction traces saved from runs for each of the ten SPECint2000 benchmarks [Spec 2000] on single-threaded cache simulators. The saved traces have a maximum length of 10 million instructions. Multi-threading workload mixes have been created using the traces from the same 10 benchmarks. (Details about the benchmarks and the simulator configurations are given later).

In addition to trace miss rate and average trace length, we used area, power, and access time, as the performance metrics for cache

implementation. Calculation of these 3 parameters is done using a readily-available cache analytical model called CACTI [Shivakumar & Jouppi 2001].

To make an overall comparison of different cache schemes, an aggregate performance index that combines the simulation and modeling results has also been used.

1.6 Organization of this Dissertation

The rest of this dissertation is organized as follows: Chapter II covers the CPC architecture and operation. Chapter III contains CPC simulation and modeling, and CPC's performance comparison with TC and BC. Chapter IV explains the use of NNM's for modeling the behavior of CPC and other caches. Finally, Chapter V presents the conclusions and suggests areas for future research.

CHAPTER II

CODE PATTERN CACHE

2.1 CPC Overview

CPC stores sequences of instructions as they execute. When the same instruction sequence is encountered later, it is fetched from CPC instead of IC. Figure 13 shows the overall block diagram of CPC connected to a superscalar processor.

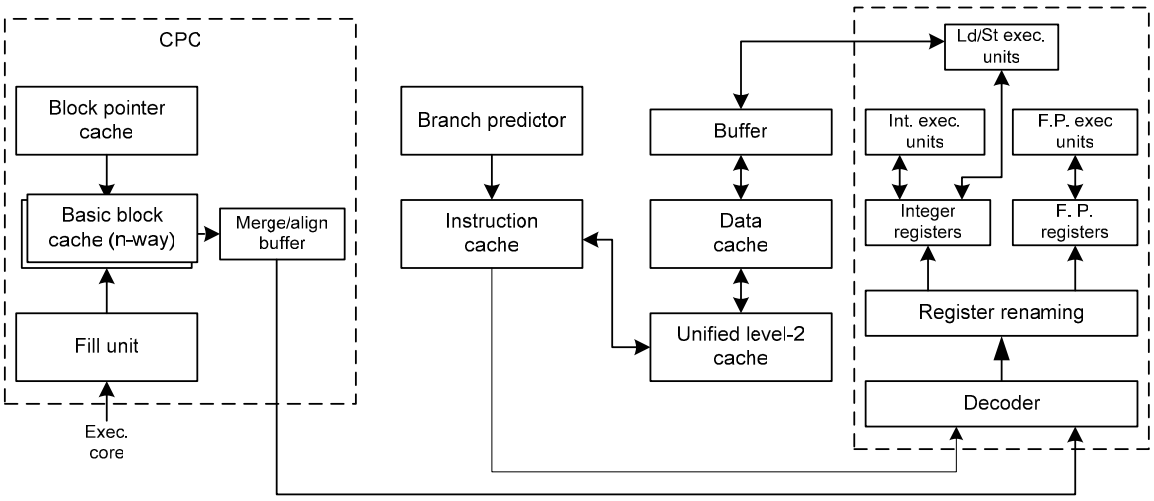

Figure 13. A superscalar processor with code pattern cache (CPC)

Unlike TC, the basic blocks that may appear in multiple traces are stored only once in CPC. The basic blocks are stored in *basic block*

cache (BBC), and the starting and ending addresses of the basic blocks are stored in a separate structure called *block pointer cache* (BPC). Each line in BPC represents a single trace by storing multiple sets of basic block (start and end) addresses. A merge-and-align buffer is used to 'assemble' a trace, before it is sent to the decoder and execution engine. The BPC lines also store the BBC way-number if BBC is configured as an n-way associative structure. Unlike BC, storage and retrieval of block sizes of varying lengths are allowed by CPC. Combined effects of variability of block sizes and set-associativity in BBC tend to lower the trace miss rate. CPC's average number of instructions stored per trace is generally higher than both TC and BC. Cache storage in CPC is more efficient than BC because the former needs replication of cache structures. TC uses only the beginning address for trace matching. Because blocks other than the beginning block are not identifiable, the trace is declared a 'miss' and a new trace build is initiated even if the required instructions are present in the trace. CPC avoids this unnecessary switching from trace utilization mode to build mode by allowing hits on intra-trace blocks; this helps improve the trace miss rate. Fixed-length BC lines may increase the chances of basic block fragmentation, i.e., the blocks straddling across multiple cache lines and leaving unused spaces at the end of the lines (instruction holes); CPC is

likely to have fewer instruction holes than TC. CPC's implementation in hardware is only slightly more complex than TC but is simpler than BC. Multiple branch predictions for end-of-block addresses are also required in a manner similar to TC and BC.

CPC is a "multiple-entry, multiple-exit" cache (Figure 14). This means that a CPC trace can start execution from any of its basic blocks instead of just the first one and exits can happen when a branch at the end of any basic block does not match the outcome of branch prediction.

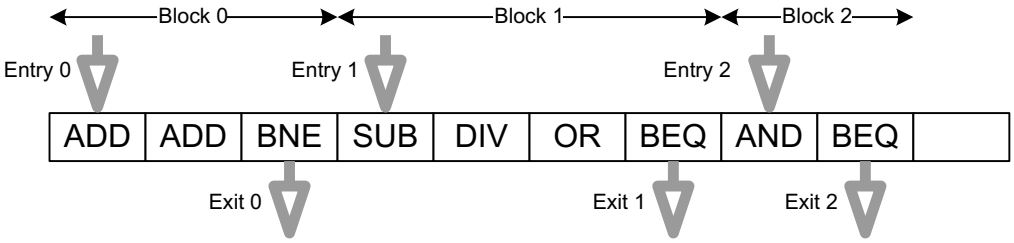

Figure 14. CPC's "multiple-entry, multiple-exit" nature: A hit to a CPC trace is possible for any of three basic blocks (Block 0, Block 1, and Block 2). So, the trace line in this example has three entry points, Entry 0, Entry 1, and Entry 2. An exit happens when any of three blocks has a mispredicted branch at its tail. Possible exit points are marked as Exit 0, Exit 1, and Exit 2.

In comparison, TC is a "single-entry, multiple-exit" cache (Figure 15) [Rotenberg et al. 1999]. In TC, the execution of a trace always starts at the beginning instruction of a trace and can terminate on any of its intra-line branches.

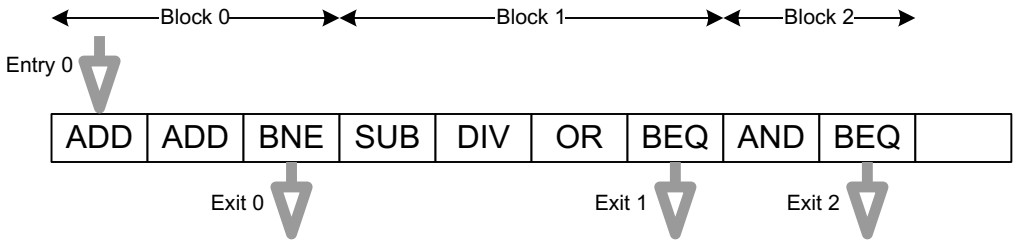

Figure 15. TC's "single-entry, multiple-exit" nature: A hit to a TC trace is possible only when the trace starting address (meaning Block 0's head address) matches. So, the trace line in this example has only one entry point, Entry 0. An exit happens when any of three blocks has a mispredicted branch at its tail. Possible exit points are marked as Exit 0, Exit 1, and Exit 2.

In the example of Figure 16, up to three blocks can be fetched every cycle.

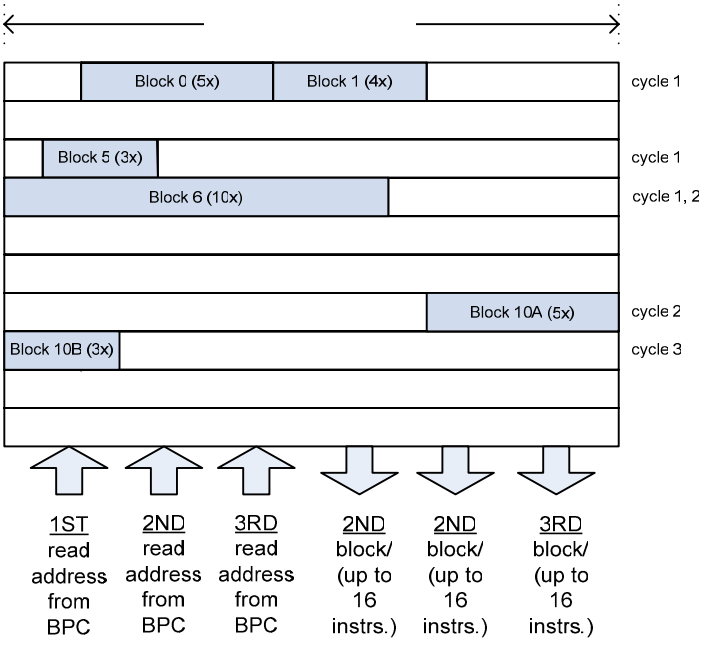

Figure 16. Basic blocks in CPC's BBC structure: up to 3 blocks can be fetched per cycle. The ability to store and fetch variable block lengths can make a CPC-trace exceed TC and BC-traces in size.

The blocks in a trace may contain a different number of instructions. The traces are allowed to exceed BBC line width. CPC is expected to have less block fragmentation than CPC.

The CPC architecture can be used both in single-threading and multi-threading modes. CPC's single-threaded version is called CPC-ST, and the multi-threaded version is called CPC-MT.

2.2 CPC-ST Architecture

An overall view of the CPC-ST architecture is shown in Figure 17.

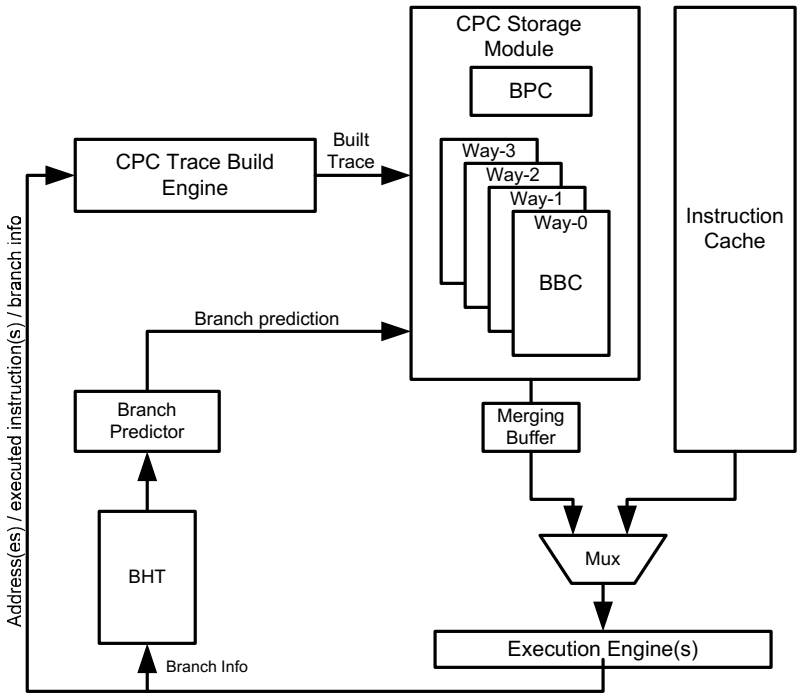

Figure 17. Overall view of the CPC-ST architecture

The building blocks of this architecture are explained in detail in the following sections.

2.2.1 Storage Module

As mentioned earlier, the CPC storage module consists of two cache structures: BPC and BBC. The full address is used for BPC lookup, whereas BBC lookup is done using tag and index fields. Interconnections of BPC and BBC are shown in Figure 18.

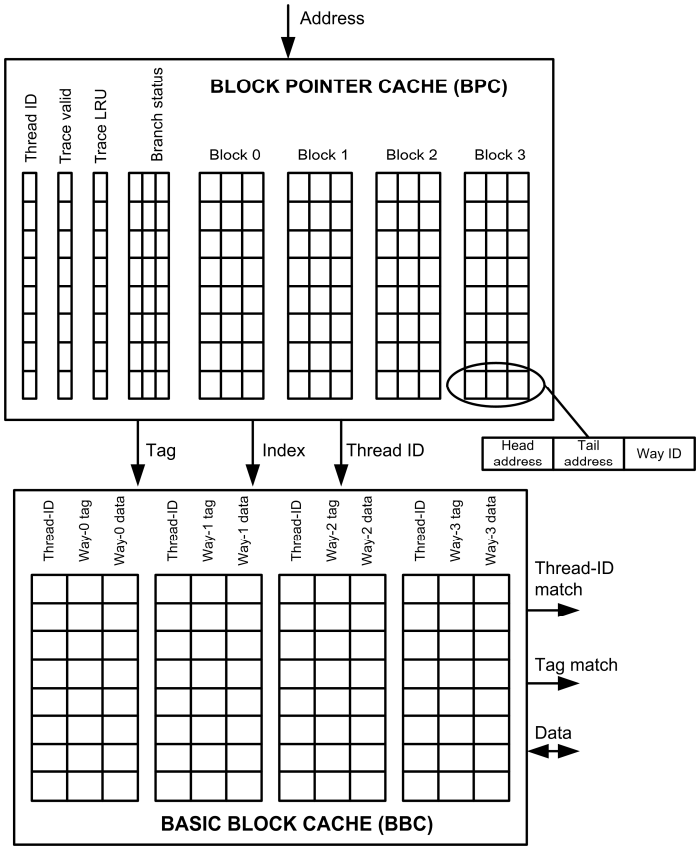

Figure 18. BPC-BBC interconnection

A single BPC-line is shown in Figure 19. Each of the lines corresponds to a single trace. BPC is made up of an array of these lines. BPC keeps track of valid basic blocks resident in the BBC. BPC starts with a state where all entries are marked *invalid*. Upon detection of a block tail, full linear addresses for both block head and block tail are placed in a BPC line. The ID's of BBC-ways where the basic blocks reside are also saved. Once all entries are populated, *conflicts* start to occur and certain lines have to be replaced. LRU fields in BPC determine which BPC

line will be evicted when there is a need to do so. *Branch status* bits store the *taken* or *not-taken* status of the branches at the ends of basic blocks. In the BPC line in Figure 19, three branch-status bits are assigned to the first 3 blocks in the trace. Branch status for the 4th block is not saved. (In the CPC-MT, the *thread-ID* field identifies which thread the trace belongs to).

Trace Valid	Head Block 0	Tail Block 0	Way ID 0	Head Block 1	Tail Block 1	Way ID 1	Head Block 2	Tail Block 2	Way ID 2	Head Block 3	Tail Block 3	Way ID 3	Branch Status	Trace LRU

Figure 19. BPC trace line. The line includes block head and tail addresses and the ID's of BBC-ways where basic blocks are stored. Other fields include thread-ID, branch status, and LRU bits.

BBC is composed of two arrays: the BBC Data Array (Figure 20) and the BBC Tag Array (Figure 21). The BBC tag array stores tags and performs tag-matching; whereas the BBC data array stores basic blocks and supplies them when needed. Basic blocks can be of any size; the sizes are only limited by the number of lines in a BBC-way.

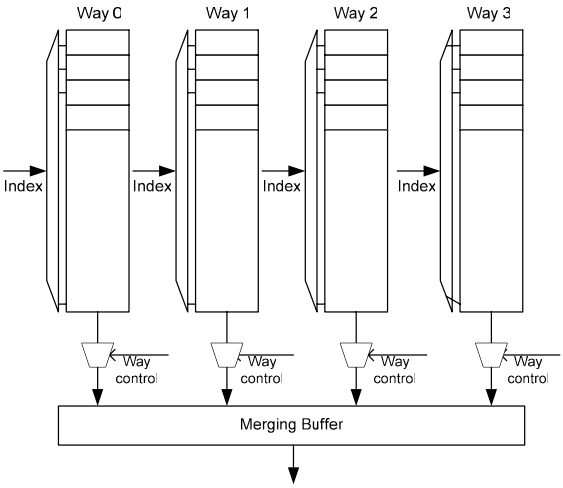

Figure 20. BBC Data Array: The array stores the basic blocks of varying lengths.

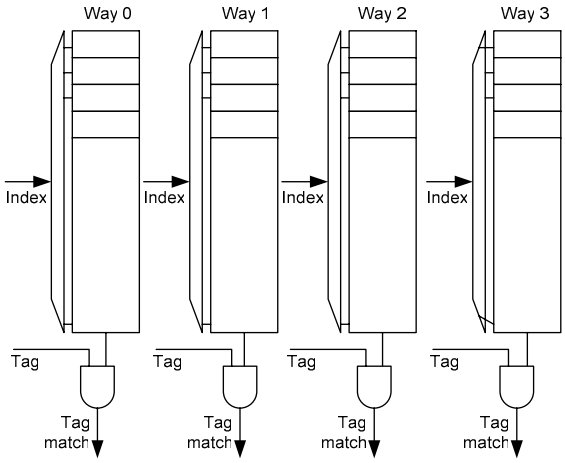

Figure 21. BBC Tag Array: Tag matching is done to determine presence of basic blocks in a BBC-way.

In order to locate a basic block in BBC, the values of index and set are derived from the block head address in BPC. The BBC-way, in which a basic block resides, is also saved in BPC. In this research, we chose 16 instructions to be the maximum number of instructions that could be fetched in one cycle from BBC. As all 16 instructions can potentially

reside in the same BBC-way, the width of the read ports on these ways has to be 16 instructions. Any traces that are longer than 16 instructions are fetched in two or more cycles.

2.2.2 Trace Build Engine

Functionally, the *trace build engine* is quite simple and primarily consists of a buffer called the *trace build buffer* (TBB) (Figure 22). While CPC is in *trace assembly mode* (explained later), the *head address* is stored in TBB, one cycle after the previous block ends. *Tail address* is the address of the control instruction that terminates the executed basic block. If a conditional branch ends the block, the *branch status* gets filled. After all TBB fields have been filled, TBB contents are copied into BPC.

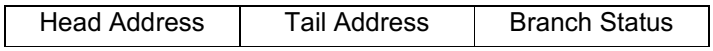

Figure 22. Trace build buffer: The buffer entry is completed upon detection of end of block condition and after the block-end branch status is known.

2.2.3 Merging Buffer

A single trace is made up of basic blocks that can be stored in one or more ways. Basic blocks read from BBC are first rearranged in the execution order and then aligned in the *merging buffer* (Figure 23) before being sent to the decoder and the execution engine. Depending on the

implementation, the merging buffer can perform its function on a single trace in one cycle.

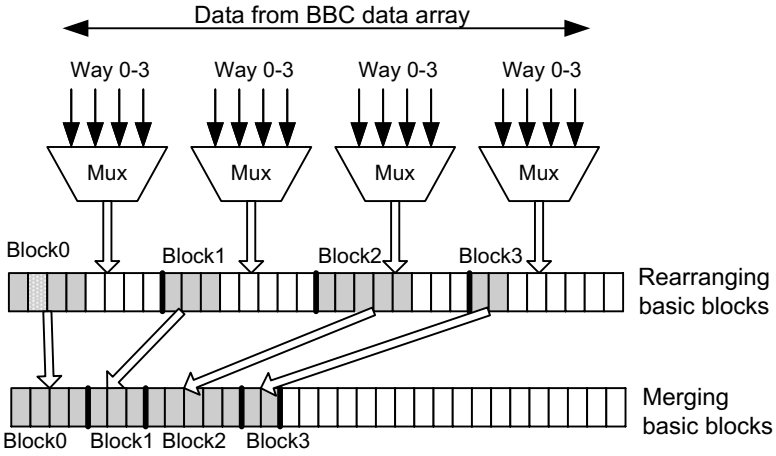

Figure 23. Merging buffer: Blocks retrieved from different BBC-ways are first re-arranged (in execution order) and aligned before being sent for execution.

2.2.4 Branch Predictor

A branch predictor is implemented in the form of a branch history table (BHT) with 2-bit counters (Figure 24). In this research, BHT size is fixed at an arbitrary value of 1024 entries. Bits 12:3 of an address are called *masked address* and are used to index into BHT. (In our addressing scheme, bits 2:0 are always zero and are not used for indexing).

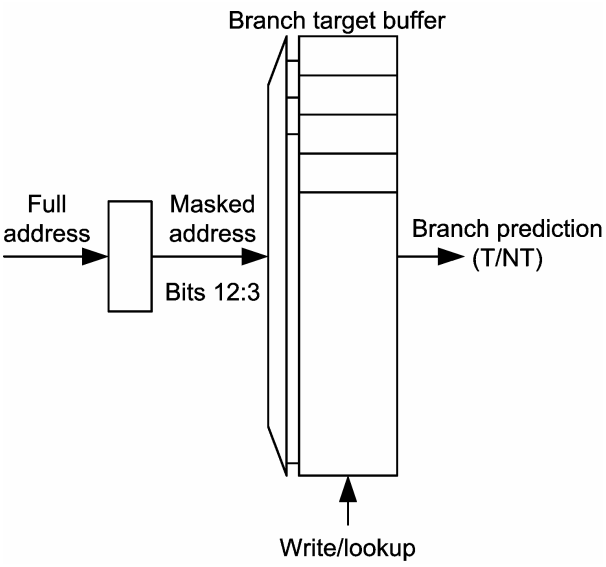

Figure 24. Branch predictor implemented in the form of a branch history table.

Two-bit saturating counters are incremented on a taken branch and decremented on a not-taken branch. A counter value of 2 or 3 predicts that a branch will be taken while a value of 0 or 1 predicts that the branch will not be taken. The branch predictor provides multiple predictions in a cycle; *n-1* predictions are made for an *n*-block trace (BPC-line).

2.3 CPC-ST Operation

A CPC-based system essentially operates in two modes: *trace assembly mode* and *trace delivery mode* (Figure 25). This means that CPC is either supplying instructions to the decoder and the execution engine or is assembling the traces for storage in BPC and BBC. While trace

assembly takes place, the instructions are fetched from IC. The logic inside a CPC-storage module is responsible for deciding CPC's operating mode.

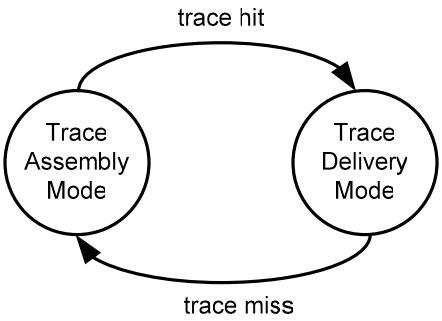

Figure 25. CPC's two modes of operation: trace assembly mode and trace delivery mode

A high-level view of functions performed during trace assembly mode and trace delivery mode is shown below (Figure 26).

A CPC trace miss can happen due to one or both of the following reasons:

(1) A trace miss occurs in BPC because the trace was not built or was overwritten by another trace, or

(2) A block miss occurs in BBC because the block was never stored or because it was over-written was another block

A CPC trace hit occurs when the following three conditions are met:

(1) The current address matches a block head address in BPC, and

(2) A tag match happens in BBC, and

(3) The branch bits (in BPC) of all hit blocks match their tail branch predictions.

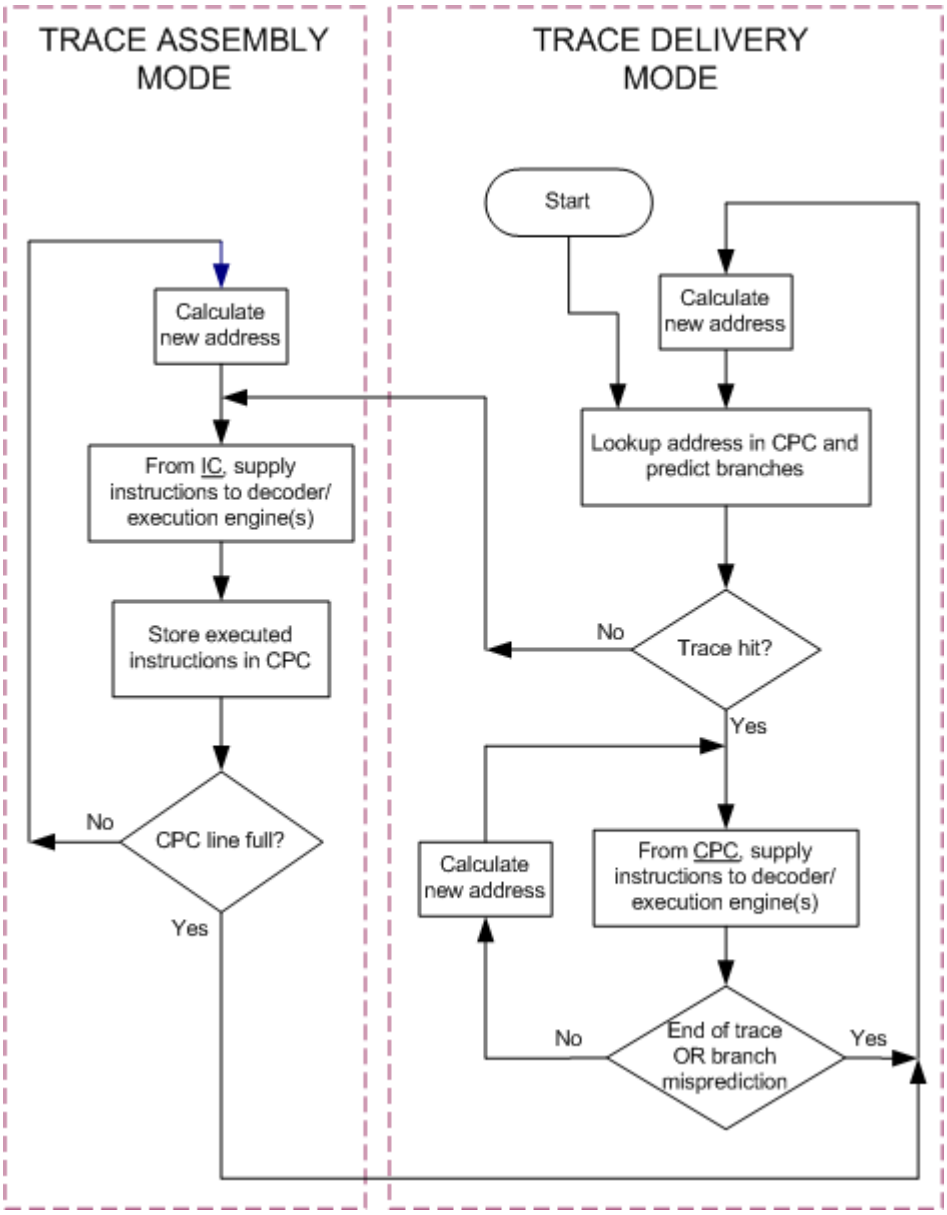

Figure 26. A high-level view of the functions performed by CPC: Tasks specific to the two operating modes are enclosed in the larger outer boxes.

2.3.1 Trace Assembly Mode

When the program initially starts running, there are misses on both CPC and IC and the instructions are fetched from the main memory. CPC does not contain any valid data at this time and CPC is in trace assembly mode.

As instructions execute, they are stored at selected locations in the BBC structure inside the CPC storage module. Concurrently, head and tail addresses of basic blocks are identified and stored in the trace build buffer (TBB) in the CPC trace build engine. After an end-of-block condition is recognized, contents of TBB (head address, tail address, and branch *taken/not-taken* status bit) are written out to a BPC line. The ID of the BBC-way in which this block's instructions are stored is also placed in BPC. After a fixed number of TBB writes to BPC line is done, the trace is considered built. In this research, four TBB writes are required to build one trace. One can note that at the beginning of an assembly process, two additional tasks are done:

(1) Finding a line in BPC for trace placement, and

(2) Finding a way for basic block placement in BBC.

When a program starts running, there are enough empty BPC lines to save the traces. However, as the program execution continues, all BPC lines become occupied and some trace needs to be evicted to make room

for a new trace. BPC-line and BBC-way replacement policies are discussed in Section 2.3.5.1 and Section 2.3.5.2, respectively.

2.3.2 Trace Delivery Mode

Upon a trace hit, CPC switches to trace delivery mode and instructions from BBC are supplied to the decoder and the execution engine. During the trace delivery mode, the LRU bits for BPC lines and BBC ways are also updated. (Refer to Sections 2.3.5.1 and 2.3.5.2 for details).

When an address matches any block head address in BPC, branch predictions are performed for the hit block and for all other blocks (except the last block) that follow the hit block. For example, if there is a hit on the 1st block in BPC, the branch-bit in BPC at the end of the block must match the branch-prediction for the block's tail address. Any mismatch of predicted branch bit with stored branch bit causes the trace to be *cut-off* at that point, which is called a *partial trace hit*. In other words, a head address is searched as the first requirement in determining a trace hit or miss; and as the second requirement, tail address of a hit block is used for looking up the branch history table for a branch prediction. A few cases are analyzed in the following two sections.

2.3.3 Branch Prediction-Related Cases

(1) Case 1:

Assume that branch status bits in BPC are *[not taken]-[not taken]-[taken]* and predicted branches are *[not taken]-[not taken]-[taken]*. In this case, all branch predictions match the branch status bits, so there is a *full-trace* hit. In BPC, the full trace contains all 4 blocks; the blocks still have to be located in BBC for a complete hit.

(2) Case 2:

Now consider a case where branch status bits in BPC are *[not taken]-[not taken]-[taken]* and predicted branches are *[not taken]-[not taken]-[not taken]*. Here, there is a misprediction on the 3rd branch, so it is considered a *partial trace* hit. The trace effectively contains only 3 blocks.

(3) Case 3:

Consider an example where there is a hit on the 2nd block in BPC and there is a branch prediction mismatch with a current block's status bit. In this case, the partial hit part is limited to only one block.

2.3.4 Miss Rate Related Cases

Basic blocks found in BBC may reside in one or more ways in a multi-way BBC configuration. The merging buffer is used to assemble these basic blocks into a full trace (as discussed in Section 2.3.1). The

method with which basic blocks are accessed from BPC and BBC is explained in the following examples.

(1) Case 1:

Assume a 4-way BBC has started executing a program and has caused a trace miss. As the code execution proceeds, the 4 basic blocks shown in Figure 27 are identified for the trace.

Block number	Block head address	Block tail address	Number of instructions	Branch status
0	0020	0030	3	Taken (T)
1	0050	0058	2	Taken (T)
2	01B8	01F8	9	Not taken (NT)
3	0200	0228	6	Don't care (X)

Figure 27. Information for a single trace that has four basic blocks of different lengths

The state of BPC after placement of the trace in BPC is shown in Figure 28. The first line shows the current trace with the *trace valid* bit set. Other traces are marked *invalid*. Storage of these basic blocks (designated by Block 0-Block 3) in BBC is shown in Figure 29.

Trace Valid	Head Block 0	Tail Block 0	Way ID 0	Head Block 1	Tail Block 1	Way ID 1	Head Block 2	Tail Block 2	Way ID 2	Head Block 3	Tail Block 3	Way ID 3	Branch Status	Trace LRU
1	0020	0030	0	0050	0058	1	01B8	01F8	2	0200	0228	2	TTN	100
0	0000	0000	0	0000	0000	0	0000	0000	0	0000	0000	0	TTT	000
0	0000	0000	0	0000	0000	0	0000	0000	0	0000	0000	0	TTT	000
0	0000	0000	0	0000	0000	0	0000	0000	0	0000	0000	0	TTT	000

Figure 28. A BPC line that contains trace information for the trace in Figure 27. Only the first BPC line contains a valid trace.

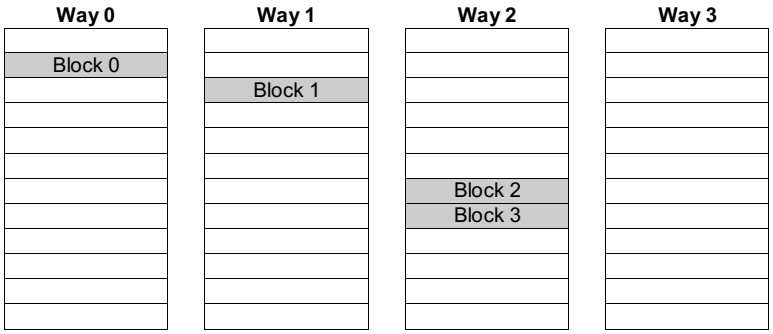

Figure 29. Placement of 4 basic blocks for a single trace in BBC: 2 basic blocks are in the same way while other two basic blocks land in their own BBC-ways.

(2) Case 2:

Assume that there are 3 valid traces (traces 0, 1, and 2) in BPC of Figure 30. Block 3 of trace 0 is common to multiple traces; the same block also represents blocks 1 and 3 of trace 2. (The common blocks are highlighted with thick borders). As mentioned previously, a block that appears repeatedly in BPC is stored only once in BBC.

Trace number	Trace Valid	Head Block 0	Tail Block 0	Way ID 0	Head Block 1	Tail Block 1	Way ID 1	Head Block 2	Tail Block 2	Way ID 2	Head Block 3	Tail Block 3	Way ID 3	Branch Status	Trace LRU
0	1	0020	0030	0	0050	0058	1	01B8	01F8	2	0200	0228	2	TTN	100
1	1	00F0	00F8	2	0310	0320	2	0050	0058	1	0390	0418	2	TNT	010
2	1	1098	10A0	3	0200	0228	2	1098	10A0	3	0200	0228	2	TTT	110
N-1	0	0000	0000	0	0000	0000	0	0000	0000	0	0000	0000	0	TTT	000
N	0	0000	0000	0	0000	0000	0	0000	0000	0	0000	0000	0	TTT	000

Figure 30. Three valid traces in BPC: There is one basic block (highlighted) that appears twice in the first trace and again in the 3rd trace.

2.3.5 Cache Replacement Policy

2.3.5.1 BPC Line Replacement

The *least recently used* (LRU) replacement scheme is used for replacing traces (lines) in BPC. The LRU bits specific to each trace are updated upon a *partial* or *full* hit of the trace. In the beginning, all LRU bits are set to 0. On a trace hit, LRU bits for all BPC lines are shifted right by a bit. LRU MSB's for all but the hit line are filled with 0; the hit-line is assigned a 1 in its LRU's MSB. When the need arises for a trace eviction from BPC, the line with the lowest LRU value is chosen. The following example further explains the workings of the BPC-LRU scheme.

One can assume an 8-line BPC with a 3-bit LRU field. At the beginning of a program, all lines in BPC LRU are set to 000. Suppose, later during the program execution, BPC LRU field values are as shown

in Figure 31 (a). (Block head, tail, and way-ID fields are omitted for simplicity). If there is a *hit* on a block in BPC-line 2 (or trace 2), LRU bits for all traces are shifted one bit to the right. The MSB of the hit block's LRU field is set to 1. Updated LRU fields are shown in Figure 31 (b). Yet another hit to the same trace changes LRU fields as given in Figure 31 (c). In Figure 31 (c), lines 0, 3, 4 and 7 have the lowest LRU value, so any one of them can be used for BPC line eviction.

BPC line	LRU bits
0	010
1	100
2	011
3	000
4	010
5	101
6	111
7	000

n hits

BPC line	LRU bits
0	001
1	010
2	101
3	000
4	001
5	010
6	011
7	000

n+1 hits

BPC line	LRU bits
0	000
1	001
2	110
3	000
4	000
5	001
6	001
7	000

n+2 hits

Figure 31. BPC LRU after *n*, *n+1*, and *n+2* hits on BPC-line 2

2.3.5.2 BBC-Way Selection & Replacement

The LRU replacement scheme has also been used for replacing basic blocks in BBC. LRU status for a BBC line is stored in its own LRU field. On the onset of code execution, all LRU bits are set to 0's. On every

access to BBC, all LRU status bits are shifted right by one bit with MSB being 0. But the line in the hit trace has its LRU's MSB set to 1.

Line/ Index	Way 0 LRU	Way 1 LRU	Way 2 LRU	Way 3 LRU
0	011	111	011	001
1	111	110	101	001
2	011	010	001	101
3	101	110	000	001
4	001	011	110	001
5	100	101	100	100
6	000	011	010	000
7	111	010	100	111

Figure 32. BBC LRU fields are 3 bits wide. Each way has its own set of LRU bits (Tag and data fields are not shown for clarity).

One can assume a 4-way BBC structure with each way composed of 8 lines. The 3-bit LRU fields are shown in Figure 32. (Tag and data fields have been omitted to simplify the figure). One can suppose a moment in time during a program run at which the LRU fields hold the values shown in the figure. Whenever there is a need to evict line-3 from BBC, Way-2 would be replaced because its LRU value is the smallest of the four.

We will look at another example to show how the LRU fields are updated when we have a block hit in BBC. For this example, LRU fields

for only one BBC-line are shown in Figure 33. The figure shows changes in LRU bits as a result of two hits on Way-3. The state of Way-3 LRU changes from “001” to “100” on first hit, and to “110” on the second hit. Note that lines other than the hit line remain unchanged.

Line/ Index	Way 0 LRU	Way 1 LRU	Way 2 LRU	Way 3 LRU	
0					
1	111	110	101	001	arbitrary state after
2					m hits

Line/ Index	Way 0 LRU	Way 1 LRU	Way 2 LRU	Way 3 LRU	
0					
1	011	011	010	100	after m+1 hits on Way 3
2					

Line/ Index	Way 0 LRU	Way 1 LRU	Way 2 LRU	Way 3 LRU	
0					
1	001	001	001	110	after m+2 hits on Way 3
2					(Way-0, 1, 2 are eviction candidates)

Figure 33. Changes in BBC LRU values after 3 hits to the same BBC-way

2.3.6 Cache Structure Indexing

Structurally, BPC is fully-associative, and BBC is n-way set associative. Looking up BPC simply involves comparing the current

instruction with the head addresses stored in BPC lines. Head and tail address fields in BPC store full addresses. This type of cache configuration has higher performance than other schemes (such as set-associative) [Rotenberg et al. 1999] but its drawback is full-address lookup instead of just the tag. This drawback, however, is not considered a major performance issue in CPC because of the limited number of trace lines being stored in BPC. Note that in BBC, new blocks can potentially overwrite the already present block(s). This clobbering of blocks can cause some performance degradation.

Indexing into BBC is demonstrated by a sample BBC configuration as follows:

(1) BBC has 4 ways

(2) Each BBC way has 512 lines

(3) Each BBC line is one instruction wide

(4) Each instruction takes up 8 bytes

(5) Addresses are 16 bits wide

(6) Instructions are 8 bytes long

For block storage in BBC, addresses are split into index and tag fields. The BBC-way length of 512 means the index field is $log_2(512) = 9$ bits wide. As the 3 least significant bits in an address are always zero, these bits do not need to be stored. The remaining 16-9-3 = 4 bits form

the tag field. Three addresses split into tag and index fields are shown in Figure 34.

Block #	Head address	Tag				Index								Ignored		
0	13E0	0	0	0	1	0	0	1	1	1	1	1	0	0	0	0
1	A7B8	1	0	1	0	0	1	1	1	1	0	1	1	1	0	0
2	C7B8	1	1	0	0	0	1	1	1	1	0	1	1	1	0	0

Figure 34. Examples of BBC addressing fields: Tag and index information for 3 blocks is shown.

Block-0 with a head address of 13E0 has an index of $(0\_0111\_1100)_2 = (124)_{10}$. So the block's starting location is the 124^{th} line in the selected BBC-way. Similarly, Block-1 with head address A7B8 is placed at location with index $0\_1111\_0111)_2 = (247)_{10}$. Note that Blocks-2 and -3 have the same index, so they occupy different BBC ways. Placement of Figure 34's blocks in BBC is shown in Figure 35.

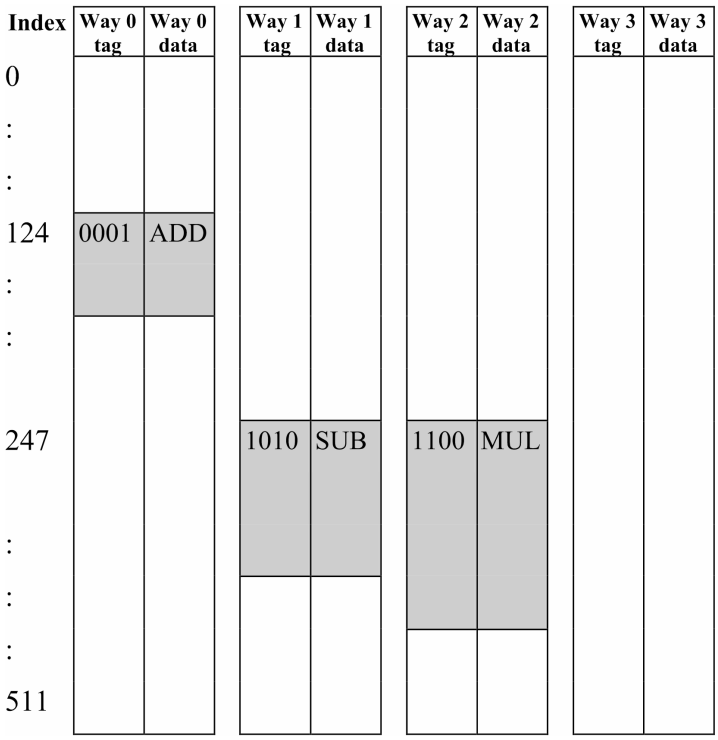

Figure 35. Block placement in BBC: The index values of Figure 34 determine block locations in BBC. Way-selection is done using the LRU bits (not shown).

2.4 CPC-MT Architecture

An overall view of CPC-MT architecture is shown in Figure 36. BPC and BBC are the only structures in CPC-MT that are shared among threads. The threads are assigned their own dedicated lines in BPC, so the threads do not overwrite each other's traces. BBC, however, is common to all threads and the basic blocks from a thread can clobber other thread's blocks.

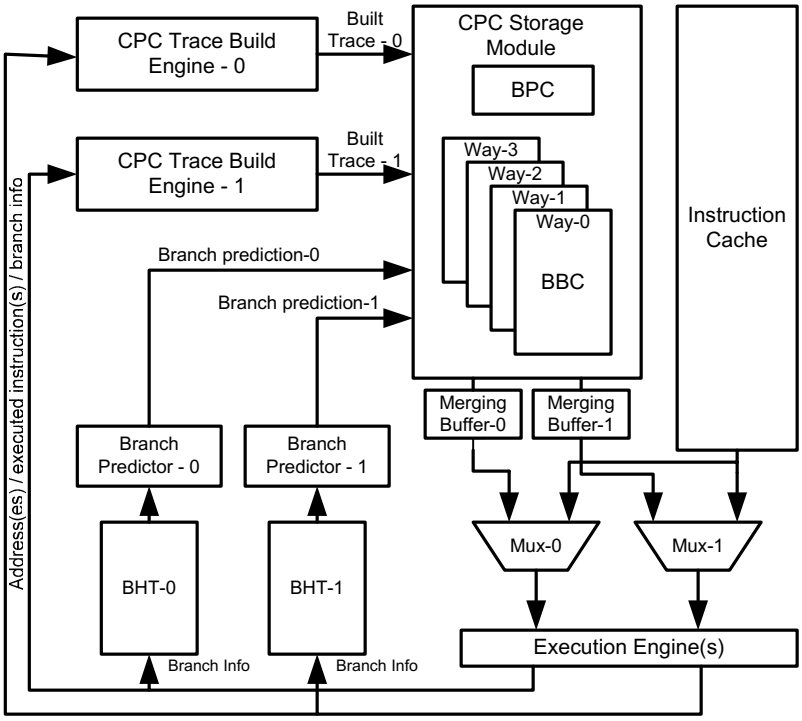

Figure 36. Overall view of a CPC-MT-based system

The following sections of CPC architecture depend on thread multiplicity:

- BPC: An additional field *thread-ID* field identifies which thread a trace belongs to
- BBC: An additional field *thread-ID* field identifies which thread a basic block belongs to
- Trace build engine: Each thread needs its own trace build engine
- Merging buffer: This buffer is also replicated for every thread
- Branch predictor: Multiple branch predictors are used, one for each thread.
- Branch history table: Every thread has its own branch history table.

2.5 CPC-MT Operation

CPC's operation in a multi-threaded mode is similar to the single-threaded mode. As mentioned earlier, the difference here is that multi-threads get the basic blocks built in their own trace build engines. Each thread also gets its own branch history table and branch predictor. The multiplicity of some resources makes trace assembly and the branch prediction process thread-independent. The CPC storage module may see simultaneous write and read requests, so the module processes them in a round-robin fashion. In our study, we allocated dedicated BPC lines to threads, but we kept BBC as a thread-shared resource. A thread-dedicated BBC configuration would have made the CPC-MT

implementation function like completely independent instantiations of CPC-ST.

CHAPTER III

CODE PATTERN CACHE SIMULATION & MODELING

3.1 CPC Simulation

In this section, we first discuss the simulators for TC, BC, and CPC. Then we cover the topics such as simulation parameters and benchmarks. After that, we go over the simulation results. Finally, we present the outcome of CPC's design space study.

3.1.1 Sim-CPC

We created a VHDL-based cache simulator called Sim-CPC to study the CPC architecture. Sim-CPC enabled functional simulation for CPC, but did not include any timing information such as cache latency. A high-level block diagram for Sim-CPC is shown in Figure 37. Every simulation cycle, one set of address and instruction is read from a benchmark's trace file until the end of the file is reached to end the simulation. After the simulation is completed, the final values of trace miss rate and average trace length are saved in a log file. We developed two more functional simulators similar to Sim-CPC: Sim-TC for TC simulation, and Sim-BC for BC simulation.

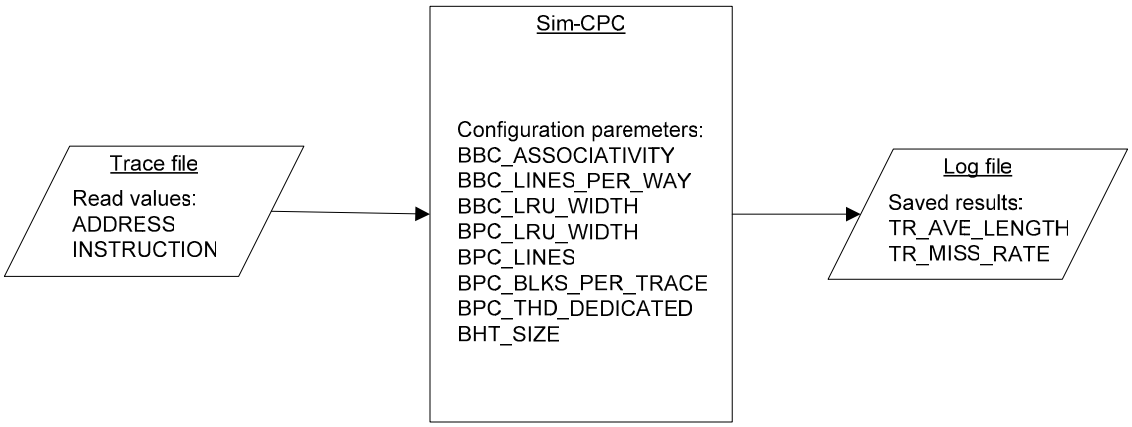

Figure 37. Sim-CPC simulator with inputs and outputs: A single set of inputs (address and instruction) is read from the trace file every cycle. At the end of the simulation, the outputs (trace miss rate and average trace length) are saved in a log file. Sim-TC and Sim-BC operate on the same principles as Sim-CPC.

We wrote a Perl script to create many variations of the three simulators (Sim-TC, Sim-BC, and Sim-CPC) using the parameters listed in Table 1. We ran all three simulators using V-System's ModelSim (version 4.f), on a Pentium-4, 2.4 GHz MS-Windows-XP-based personal computer.

Table 1. Configuration parameters for Sim-TC, Sim-BC, and Sim-CPC

Parameter	TC	BC	CPC
Number of lines in BPC	N/A	64, 256, 512	64, 256, 512
Max number of traces	64, 256, 512	64, 256, 512	64, 256, 512
Number of ways in TC/BPC	1	1	1
Number of lines in each BBC way	N/A	512, 1024, 2048	512, 1024, 2048
Cache capacity (KB)	1K, 2K, 4K, 8K, 16K	1K, 2K, 4K, 8K, 16K	1K, 2K, 4K, 8K, 16K
TC/BBC associativity	1 way (direct)	1 way (direct)	1-way (direct), 2-way, 4-way
Number of threads	1, 2, 4, 8, 16	1, 2, 4, 8, 16	1, 2, 4, 8, 16
Max basic blocks per trace	4	4	4
Max possible number of instructions per trace	16	16	Not limited
Max number of instructions delivered per cycle	16	16	16
Entries in branch history table	1024	1024	1024

A sample ModelSim simulation screen for Sim-CPC is shown in Figure 38. A new address-instruction set (*addr* and *instr*) is read every *clk* cycle.

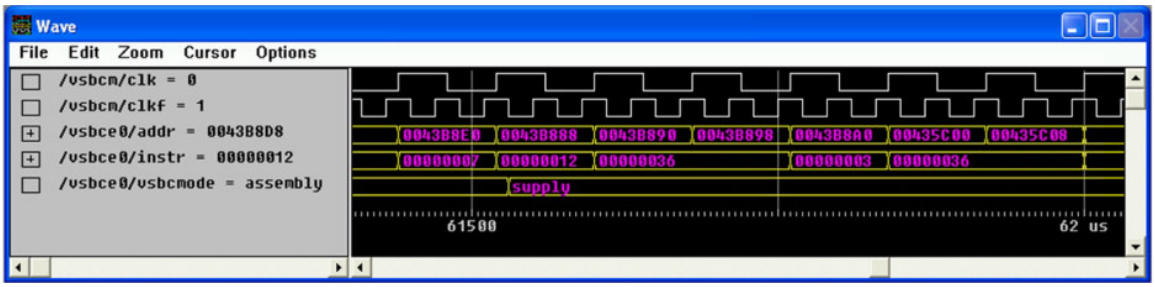

Figure 38. Sim-CPC simulation using ModelSim: An address (*addr)* and an instruction (*instr)* are read from the benchmark trace file every *clk* cycle. A trace hit causes operation-mode switch from trace assembly to trace delivery (supply) at 61530 ns.

3.1.2 Benchmark Programs

For performance comparison of the caches, we used 10 benchmark programs (listed in Table 2) from the SPECint2000 suite [Spec 2000]. The programs were compiled with *gcc* compiler (version 2.7.2.2 using -O0 option). Refer to Appendix for a detailed description of the 10 benchmarks.

Table 2. Benchmarks for comparing CPC with TC and BC

Benchmark	**Description**	**Input Data Set**
bzip	Compression	input.random
crafty	Game playing: chess	crafty.in
gap	Group theory, interpreter	test.in
gcc	C language compiler	cccp.i
gzip	Compression	input.compressed
mcf	Combinatorial optimization	inp.in
parser	Word processing	test.in
perlbmk	PERL language	test.pl, test.in
vortex	Object-oriented database	lendian.raw
vpr	FPGA circuit placement & routing	net.in, arch.in

3.1.3 Workload Mixes

Using the integer benchmarks of Table 2, we created ST and MT-workloads (Table 3). WL0a-WL0j are single-threaded workloads and WL1-WL9 are multi-threaded workload mixes. Note that for the 16-thread configuration, some benchmarks were run on more than one thread.

Table 3. Integer workload mixes for single and multi-threaded simulations

Workload/ Mix #	Thread Count	Benchmarks
WL0a-WL0j	1	bzip, crafty, gap, gcc, gzip, mcf, parser, perlbmk, vortex, vpr
WL1	2	bzip, crafty
WL2	2	gap, gcc
WL3	2	parser, perlbmk
WL4	2	vortex, vpr
WL5	4	bzip, crafty, gap, gcc
WL6	4	gap, gcc, gzip, mcf
WL7	8	bzip, crafty, gap, gcc, gzip, mcf, parser, perlbmk
WL8	8	gap, gcc, gzip, mcf, parser, perlbmk, vortex, vpr
WL9	16	bzip, crafty, gap, gcc, gzip, mcf, parser, perlbmk, gap, gcc, gzip, mcf, parser, perlbmk, vortex, vpr

3.1.4 Simulation Results

We ran simulations for different configurations of TC, BC, and CPC (already described in Section 3.1.1) to collect the performance data. In order to make reasonable comparisons of TC with BC and CPC, we simulated similar sizes of caches. For example, a CPC (BBC) of 1K capacity was compared with the TC of 1K capacity and with the BC (block cache) of 1K capacity. We ran simulations for 1K, 2K, 4K, 8K and 16K caches, in single-way configurations.

The ST notations (for example, in Figure 39) can be understood with these two examples: “bzip 1K” represents the miss rate or trace length comparison for *bzip* benchmark when run on a 1K cache; “crafty

8K" represents the miss rate comparison for *crafty* benchmark when run on an 8K cache. Similarly, the MT notations (for example, in Figure 40) can be explained with these two examples: "WL1_2thd_1K" stands for the relative miss rate or trace length when a WL1 (2-thread) workload is run on a 1K cache, and "WL7_8thd_8K" stands for the relative miss rate or trace length for a WL7 (8-thread) workload when run on an 8K cache. The same (ST and MT) notations will be used through out Section 3.1.4.

3.1.4.1 Miss Rates in Single-Threaded Environment

For ST-workloads (WL0a-WL0j in Table 3), the trace miss rates are shown in Figure 39. In the ST environment, CPC's miss rate reduction over TC varied from 43% to 95%, whereas CPC's miss rate reduction compared to BC was between 5% and 48%. The miss rate reduction percentages dropped slightly when cache sizes were increased. As cache sizes grew, the gap between CPC and BC miss rates was smaller than the gap between CPC and TC miss rates. CPC's miss rates for larger-block benchmarks (e.g., *crafty, gcc*, *gzip, perlbmk*) seem to be better than smaller-block benchmarks.

CPC's miss rate gains over TC can be attributed to the reduction in the block overlap among the CPC traces. Being able to hold blocks that are longer than what fixed-width BC would store made it possible for CPC to have lower miss rates than BC. CPC, with 1K trace capacity, has

miss rates comparable to 16K TC or to 8K BC. However, if we keep increasing TC and BC's cache capacity, their performance gap with CPC will start to shrink. Hossain (2002) suggested using 98% or higher accuracy of branch prediction to utilize the full potential of TC. The same recommendation could also improve CPC's performance.

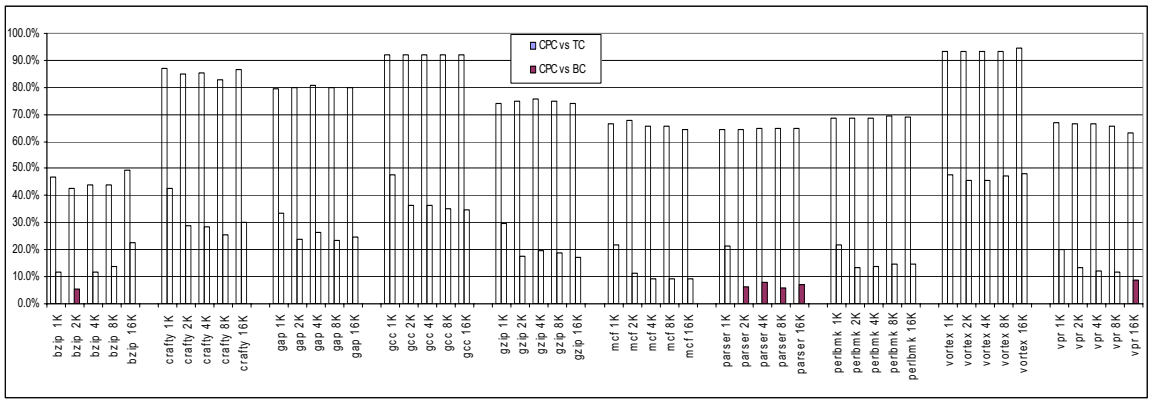

Figure 39. CPC's miss rate comparison with TC and BC in single-threading environment. On average, CPC is 73.7% better than TC and 22.7% better than BC.

3.1.4.2 Miss Rates in Multi-Threaded Environment

The miss rate comparisons for MT-workloads (WL1-WL9 in Table 3) are shown in Figure 40. With these workloads, CPC consistently performed better than TC, with trace miss rate improvements ranging from 69% to 95%. CPC had somewhat similar miss rates as BC for WL2 and WL3 workload mixes; whereas for other 8 workloads, CPC's miss rate was much better than BC.

CPC-MT's miss rate gains over TC-MT can be ascribed to reduction in overlapping instructions among the traces. Similarly, CPC's accommodation of variable length blocks, as compared to BC's fixed length blocks, seems to have helped CPC offer better miss rates than BC.

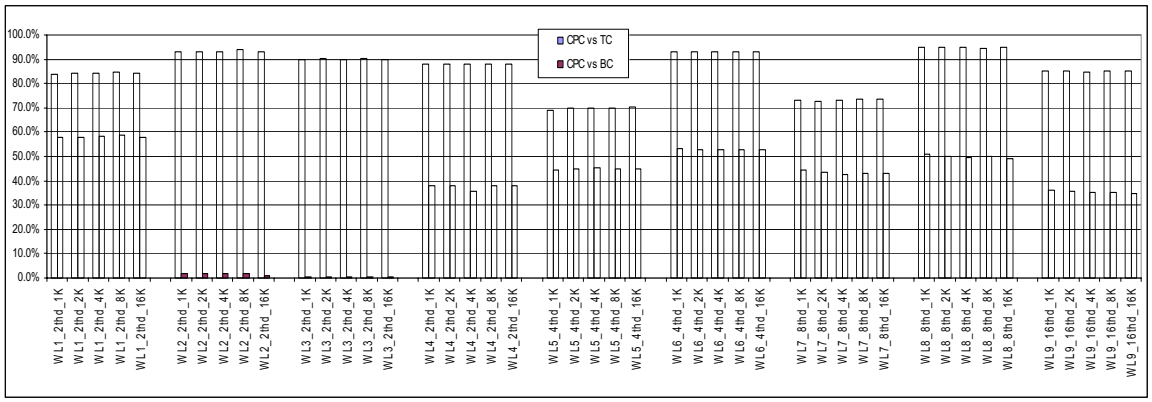

Figure 40. CPC's miss rate comparison with TC and BC in multi-threading environment. On average, CPC is 85.7% better than TC and 36% better than BC.

3.1.4.3 Trace Length in Single-Threaded Environment

For single-threaded workloads (WL0a-WL0j in Table 3), the trace length comparisons are shown in Figure 41. Trace length gains varied widely in the ST-environment. TC's trace lengths ranged from -10% to 7% of the CPC traces for five of the workloads; for the other five workloads, the trace length gains of CPC were up to 254% of the TC trace lengths. CPC traces were up to 265% longer than BC's traces. CPC's ability to hold the blocks that are not length-limited seems to be the reason for the higher value of trace lengths.

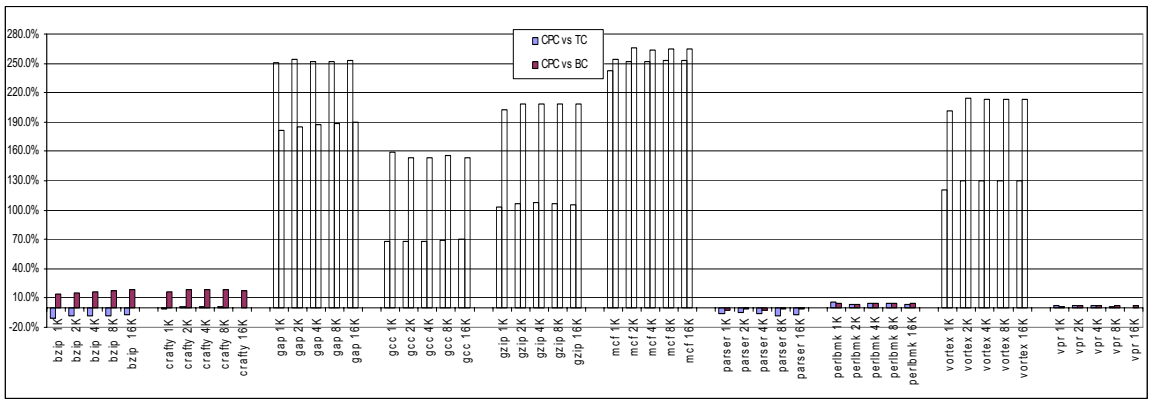

Figure 41. CPC's trace length comparison with TC and BC in single-threading environment. On average, CPC is 79.7% better than TC and 106.1% better than BC.

3.1.4.4 Trace Length in Multi-Threaded Environment

Trace length comparisons for MT workloads (WL1-WL9 in Table 3) are shown in Figure 42. CPC's trace length improvement over TC ranged from -3% to 293%; improvements over BC were from -4% to 315% (Figure 42). While multi-threading, BPC is equally divided among the threads. For example, for dual threads, half the BPC lines are dedicated to one thread and the other half to the other thread. On the other hand, all BBC lines are open to all threads, which can cause the traces from different threads to clobber each other. The combination of reduced BPC capacity per thread and inter-thread trace collisions are the apparent reasons for a wide variation of performance results while multi-threading.

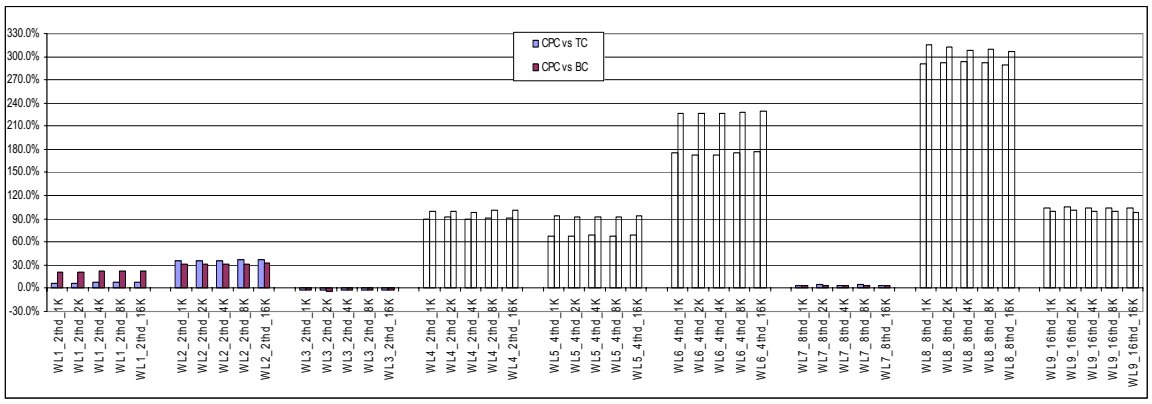

Figure 42. CPC's trace length rate comparison with TC and BC in multi-threading environment. On average, CPC is 86.1% better than TC and 98.4% better than BC.

3.1.4.5 CPC's Overall Gains in Trace Miss Rate and Trace Length

The average values of trace miss rates and trace lengths are shown in Table 4 and Table 5. In all cases, CPC's average values of miss rate and trace length are better than those of TC and BC.

Table 4. Miss rate comparison for single and multi-threaded environments

	Average TC miss rate	**Average BC miss rate**	**Average CPC miss rate**
Single-threaded workload (WL0a-WL0j)	15.6%	7.7%	4.4%
Multi-threaded workload (WL1-WL9)	45.8%	21.8%	5.9%

Table 5. Trace length comparison for single and multi-threaded environments

	Average TC trace length	Average BC trace length	Average CPC trace length
Single-threaded workload (WL0a-WL0j)	12.5	11.1	24.3
Multi-threaded workload (WL1-WL9)	12.6	11.9	24.0

3.1.4.6 Design Space Study

As the subject of this research is CPC itself, we conducted additional simulations to study CPC's own design space. We explored sensitivity of CPC's performance to BPC size, BBC size, and thread count.

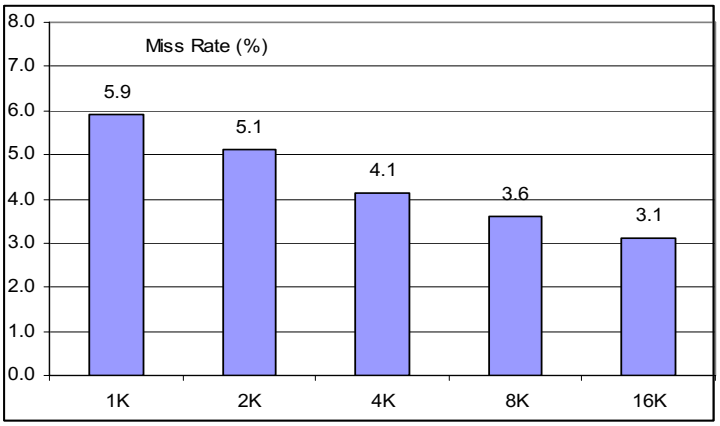

Figure 43: Effect of varying CPC cache (BPC) size (shown on horizontal axis) on miss rate: A drop in miss rate happens with increase in BPC capacity.

As expected, the results (Figure 43) showed us that increase in BPC size improved the miss rate. The average trace lengths, however,

remained nearly unaffected by the size variation (Figure 44). Currently, even a single block hit (partial hit) is considered a trace hit. Changing the definition of partial hits to two or more blocks may result in higher averages of trace lengths; although this redefinition of partial hits may reduce the trace miss rate.

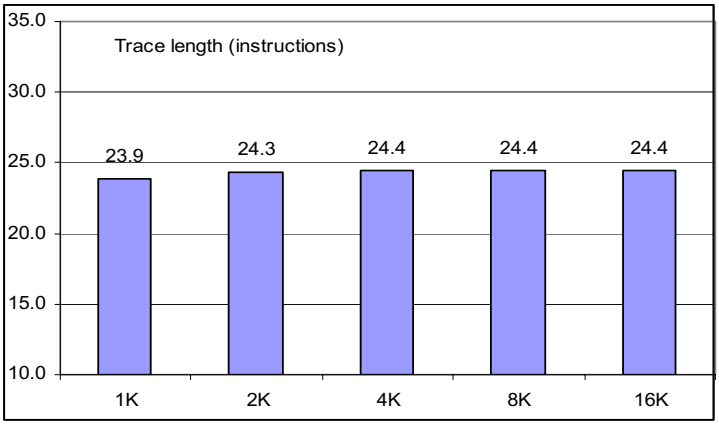

Figure 44. Effect of varying CPC cache (BPC) size (shown on horizontal axis) on trace length. The trace length is relatively insensitive to cache size.

We observed that increasing BBC associativity from 1 to 2 had the largest miss rate improvement, but the gains flattened out with higher associativities of 4 and 8 (Figure 45); the trend resembles the familiar cache-associativity curve.

Figure 45. Effect of varying CPC-BBC associativity on miss rate: After an initial drop in miss rate, it flattens out with increase in associativity. (Horizontal axis shows number of BBC-ways.)

The trace lengths were not affected significantly by changing the BBC-associativity (Figure 46). The reason for this may be that the storage of a basic block is not spread over multiple ways, so the higher associativity does not help increase the trace lengths.

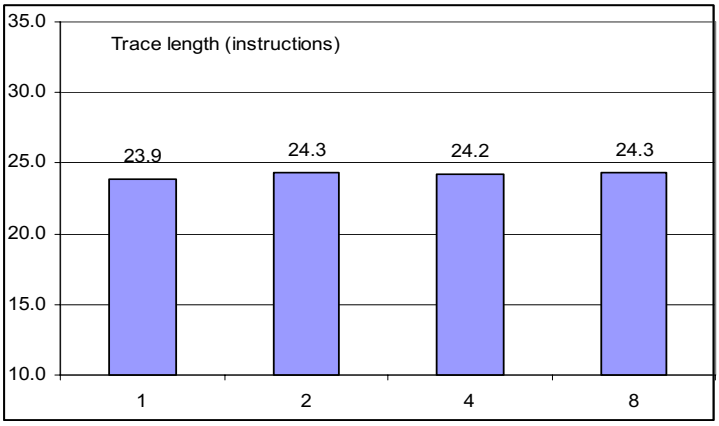

Figure 46. Effect of varying CPC-BBC associativity of trace length: The trace lengths are not affected very noticeably with the change in BBC-associativity. (Horizontal axis shows number of BBC-ways.)

Results for the sensitivity of miss rate (Figure 47) and trace length's sensitivity to thread count (Figure 48) did not exhibit a consistent upward or downward pattern which may be because multiple threads can change the locality of reference in BBC, a shared-memory structure. A future in-depth study of: (a) benchmarks' branch behavior (type, taken/not-taken frequency, etc.) and (b) inter-thread clobbering effect, may help improve our understanding of the performance of CPC in multi-threading environments.

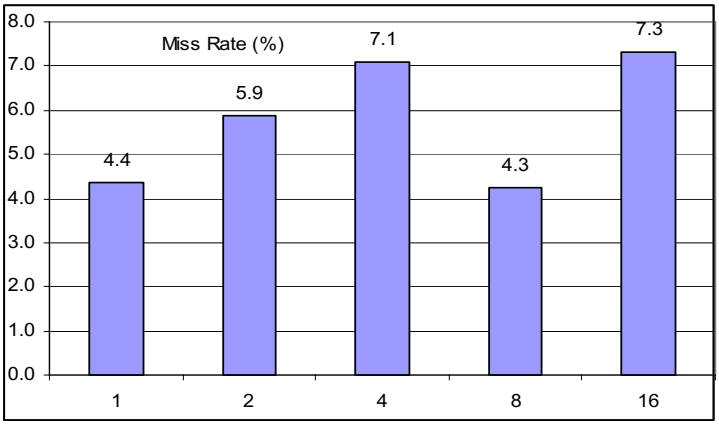

Figure 47. Effect of varying thread count on miss rate: Miss rates do not seem to have a consistent correlation with the thread count. (Horizontal axis represents thread-count.)

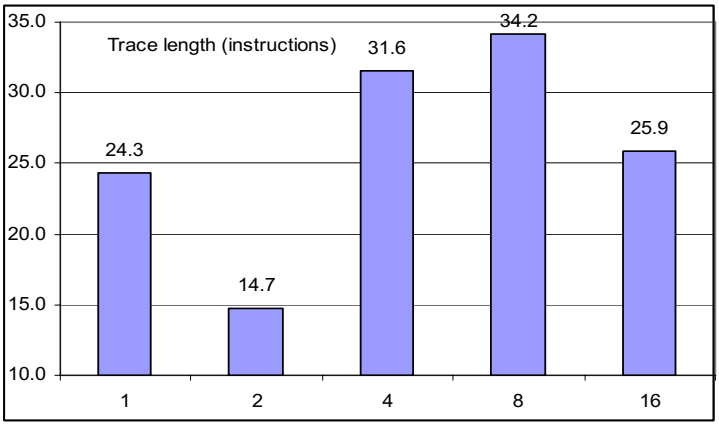

Figure 48. Effect of varying thread count on trace length: No clear relationship between thread count and trace lengths is visible. (Horizontal axis represents thread-count.)

3.2 CPC Modeling

In this section, we start with a brief introduction to CACTI, a tool for modeling cache power, area, and access time. Then, we cover CACTI's application to the study of trace and block-caches. And finally, we discuss the results from CACTI runs.

3.2.1 CACTI

For design optimization, it is helpful to quantify the relationship between the cache configuration factors, such as cache size, cache associativity, and block width. The physical parameters such as *aspect ratio* and *sub-blocking* are also important [Shivakumar & Jouppi 2001]. (Aspect ratio is calculated by dividing cache's total height by width; sub-blocking means division of the cache into independent banks to allow simultaneous, multiple accesses). Another important metric for cache performance is the access time. Wada's [Wada et al. 1992] cache access-time model described an analytical method for studying design space without the need for time-consuming SPICE simulations. Amrutur & Horowitz (2000) presented models for analyzing access time, power, and area for SRAM's. The CACTI tool [Shivakumar & Jouppi 2001] provides means for more comprehensive cache studies and allows integration of models for access time (cycle time), area, aspect ratio, and power. Bringing together these models provides an efficient way of closing in on

design configurations that are reciprocally consistent. Hanson et al. (2003) used CACTI to conduct their study of reduced static energy consumption in on-chip level-1 and level-2 caches.

3.2.2 Using CACTI

This dissertation utilizes the CACTI (version 3.0) model for comparing the access time, consumed power, and die area for TC, BC, and CPC. To make the comparison meaningful, storage capacities of cache structures in TC, BC, and CPC were kept the same. For example, a 32K TC was compared with 32K BC and 32K CPC. One can note that the equivalents of BC's trace table and CPC's BPC do not exist in TC; this observation has to be considered while comparing TC's power and area with BC and CPC. Both BC and CPC have additional structures used for merging and re-arranging the blocks fetched from the block cache (in BC) and BBC (in CPC). The implementation of these merging structures uses many fewer transistors than the caches themselves, so they are currently being ignored. Cache capacities of 1K, 2K, 4K, 8K, 16K, and 32K were used in comparing all three cache schemes. In this chapter, we use the CACTI model parameters that are listed in Table 6. A point to note is that for a similar amount of trace capacity, CPC's BPC is twice in size as the trace table in BC; CPC stores both

block-head and tail addresses, while BC only stores head addresses. (In this dissertation, the BC design is assumed to have no renaming table).

Table 6. CACTI model parameters for TC, BC, and CPC

Parameter	TC	BC	CPC
Technology (um) *	0.18	0.18	0.18
RWP *	1	1	1
RP *	1	1	1
WP *	1	1	1
NBanks *	1	1	1
Associativity *	1	1	1
BBC (or equivalent cache) line size (bytes)	64	4 x 64	64
Number of lines/traces	128	128	128
BPC (or equivalent cache) size (KB)	N/A	2K	4K
BBC (or equivalent cache) size (KB)	1K, 2K, 4K, 8K, 16K, 32K	4 x (1K, 2K, 4K, 8K, 16K, 32K)	1K, 2K, 4K, 8K, 16K, 32K

* Refer to [Shivakumar & Jouppi 2001] for detailed explanations of CACTI parameters.

3.2.3 Modeling Results

3.2.3.1 Access Time

Once can note that BC access requires a search for an address in the trace table followed by the actual basic block lookup in the block cache. So the total access time is the sum of the two accesses. CPC accesses both BPC and BBC structures in parallel, so CPC's access time is the longer of the two access times for BPC and BBC. The time calculations are done using these equations:

$$\text{TC access time} = \text{TC access time} \quad \{1\}$$

$$\text{BC access time} = \sum (\text{trace table access time, block cache access time}) \quad \{2\}$$

$$\text{CPC access time} = \max (\text{BPC access time, BBC access time}) \quad \{3\}$$

The access time of all three caches increases as the cache size is increased (Figure 49). TC and CPC have the same access times, which is lower than that of BC. With an increase in cache size, BC access time increases at a faster rate than CPC and TC because of the sequential access of BC's larger trace table and block cache structures.

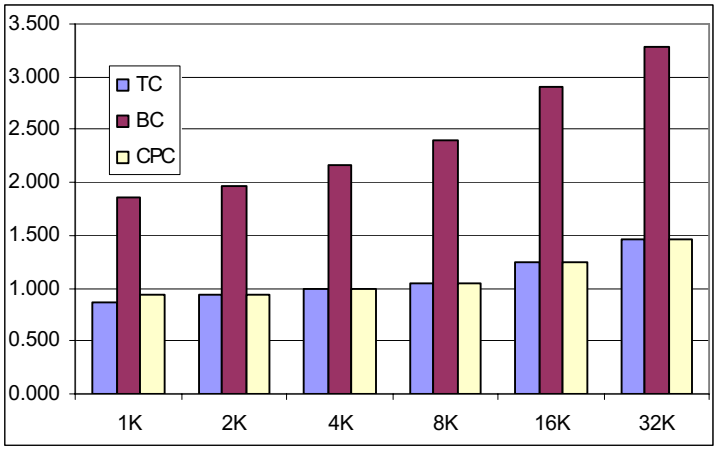

Figure 49. Access time (ns) comparison for TC, BC, and CPC

3.2.3.2 Consumed Power

As we pointed out in earlier discussions, TC had only one cache structure, while BC and CPC had two such structures. So, the power calculations for the three types of caches are done as follows:

$$\text{TC power} = \text{TC cache power} \quad \{4\}$$

$$\text{BC power} = \sum (\text{trace table power, block cache power}) \quad \{5\}$$

$$\text{CPC power} = \sum (\text{BPC power, BBC power}) \quad \{6\}$$

Power consumption graphs (in Figure 50) show that the power consumption rises as cache sizes increase. BC's power, however, increases at a higher rate than TC and CPC.

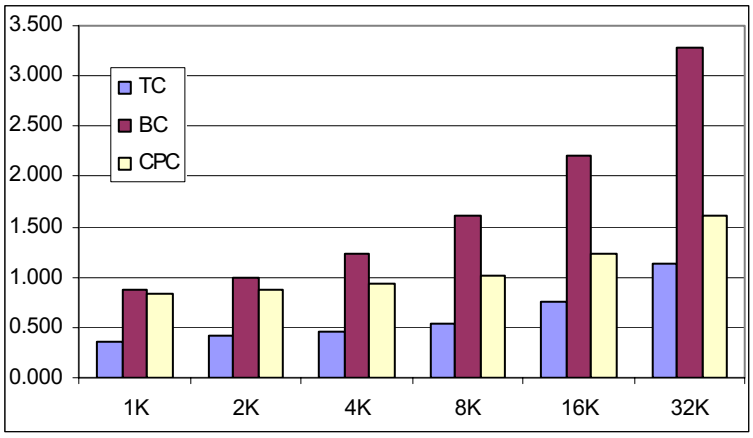

Figure 50. Power comparison (nJ) for TC, BC, and CPC

Multiplicity of block cache structures in BC is the reason for the marked difference in the power consumption among BC and other two cache schemes.

3.2.3.3 Area

Just like the power calculations, the additional areas of BC's trace table and CPC's BPC have to be taken into account when calculating the die area. So the area calculations are performed as follows:

$$\text{TC area} = \text{TC cache area} \quad \{7\}$$

$$\text{BC area} = \sum (\text{trace table area, block cache area}) \quad \{8\}$$

$$\text{CPC area} = \sum (\text{BPC area, BBC area}) \quad \{9\}$$

As we can see in Figure 51, the area for TC and CPC increases somewhat linearly, while the total area for BC increases at a faster rate. The redundancy of block cache in BC contributes significantly to the additional area.

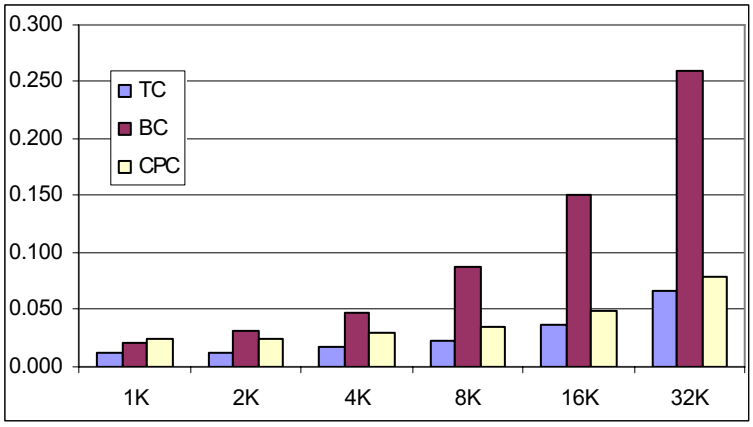

Figure 51. Area comparison (cm^2) for TC, BC, and CPC

3.3 Combining Simulation and Modeling Results

So far, we have studied simulation and modeling outcomes for TC, BC, and CPC, separately. But in order to compare the three caches in a mutually consistent manner, we need a single performance measure. So we introduce a metric called *aggregate performance index* (API) that brings together the simulation and modeling results. Equal weights are assigned to all parameters that make up the API. The API is defined by the following equation:

$$\begin{aligned}\text{API} = {} & (\%\ \text{miss rate gain}) + (\%\ \text{trace length gain}) + (\%\ \text{power gain}) \\ & + (\%\ \text{area gain}) + (\%\ \text{access time gain})\end{aligned} \quad \{10\}$$

How CPC compares with the same-sized TC is shown in Table 7. Similarly, CPC's comparison with BC is given in Table 8. Although CPC consumes more power and area than TC, the lower miss rate and longer trace lengths give CPC an overall lead over TC. API for TC increases as the cache size increases, which is due to the relative reduction in CPC's area and power when the cache size increases. Redundant cache structures in BC prove to be a significant disadvantage when power and area comparisons are made with CPC, so CPC's API figures are considerably better than BC. Here again, CPC's API is higher for larger cache sizes.

On average, CPC has 62% higher API than TC, and 254% higher API than BC. We can therefore say that CPC is an overall better cache scheme than TC and BC.

Table 7. Aggregate performance index (API) for different cache sizes - CPC vs. TC

Cache size	Miss rate gain	Trace length gain	Power gain	Area gain	Time gain	API
1KB	74.0%	77.5%	-81.8%	-17.6%	0.0%	52.1%
2KB	73.6%	80.3%	-76.9%	-23.6%	0.0%	53.3%
4KB	73.7%	80.3%	-72.8%	-24.7%	0.0%	56.6%
8KB	73.2%	80.1%	-66.7%	-21.8%	0.0%	64.9%
16KB	73.8%	80.1%	-54.1%	-16.8%	0.0%	82.9%

Table 8. Aggregate performance index (API) for different cache sizes - CPC vs. BC

Cache size	Miss rate gain	Trace length gain	Power gain	Area gain	Time gain	API
1KB	29.8%	103.3%	11.4%	-5.5%	48.1%	187.0%
2KB	20.2%	106.7%	17.5%	43.6%	47.8%	235.9%
4KB	21.1%	106.5%	30.2%	44.9%	52.0%	254.7%
8KB	20.6%	107.2%	40.6%	68.3%	55.3%	292.0%
16KB	21.8%	107.1%	45.7%	70.3%	56.6%	301.5%

CHAPTER IV

NEURAL NETWORK MODELS FOR CACHES

4.1 Neural Networks

Neural networks (NN's) mimic the ability of a human brain to find patterns and uncover hidden and relationships in data. NN's are more effective than statistical techniques for organizing data and predicting results and are very efficient in modeling non-linear systems. In general, substantially fewer resources and time are required to build an NNM when compared to a mathematical model [Caudill 1990], [Uhrig 1995], [Yale 1997].

4.1.1 Processing Elements

A neural network (NN) is defined as a computational system comprised of simple but highly interconnected *processing elements* (PE's) (Refer to Figure 52 and Figure 56) [Stegmayer & Chiotti 2004]. PE's are neural network equivalents of biological neurons. Similarly, neural network interconnections are equivalents of synapses that connect a neuron to others. Information is processed by the PE's by dynamically responding to their inputs. Unlike conventional computers that process

instruction and data stored in the memory in a sequential manner, the NN's produce outputs based on a weighted sum of all inputs in a parallel fashion [Caudill 1990].

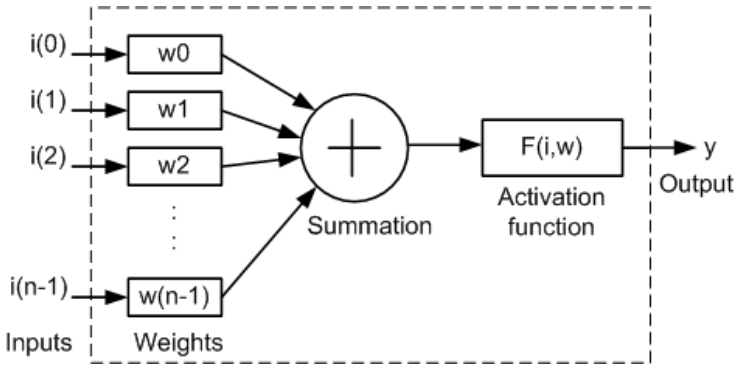

Figure 52. Processing element – building block of a neural network

In Figure 52, the inputs (*i(0..n-1)*) to a PE are scaled with weights (*w(0..n-1)*) and summed before being passed through an *activation function*. The activation function determines whether a PE *fires* or not. The input-output relationship of an activation function may be linear or non-linear. Two linear functions, *linear* and *ramp*, are shown in Figure 53 and Figure 54, with outputs in the [-1, 1] range.

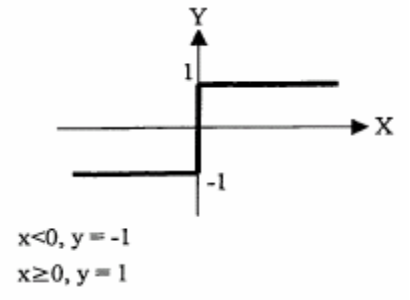

Figure 53. A step activation function

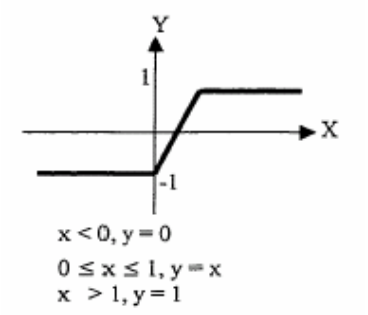

Figure 54. A ramp activation function

A s*igmoid* (non-linear) activation function has an s-shaped output between the limits [0, 1]. The function is defined as follows and is shown in Figure 55 [UTexas 2005]:

$$y = \frac{1}{(1+e^{-x})} \qquad \{11\}$$

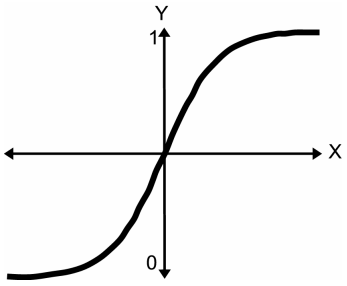

Figure 55. A sigmoid activation function

4.1.2 A 3-Layer NN Topology

Figure 56 shows the topology of a simple *feed-forward* NN with 4 inputs (*i(0)..i(3)*) and 2 outputs (*y(0)..y(1)*). The NN has 4 input neurons (*PE(i,0)..PE(i,3)*), one hidden layer with 5 neurons (*PE(h,0)..PE(h,4)*), and one output layer (*PE(o,0)..PE(o,1)*) with 2 neurons [Caudill 1990].

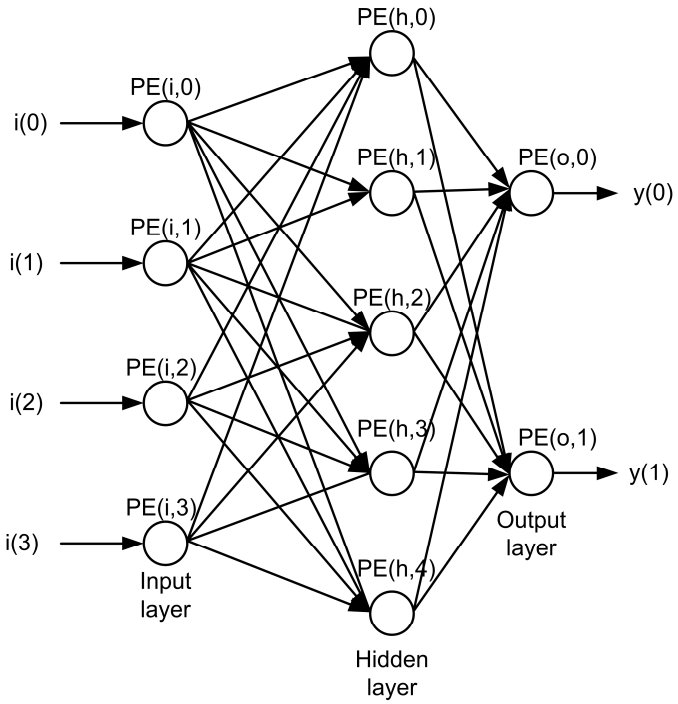

Figure 56. Topology of a 3-Layer Feed-Forward Neural Network

Each input of an NN corresponds to a single attribute, such as the cache size, cache associativity, etc. The type of inputs determine whether their values are *discrete* (e.g., 1 or 0 representing *yes* or *no*, *true* or *false* values) or *continuous* (e.g., 0.27 representing cache miss rate). The

output of an NN is the prediction we are trying to make. Just like inputs, the outputs can be discrete or continuous.

4.1.3 Learning Mechanism

The *weights* correspond to the relative strength (or numerical values) assigned to the NN inputs or the connections that transfer data from one neuron layer to the other. Iterative adjustments of weights make NN's *learn.* NN's use different types of *learning* (or *training*) mechanisms, the most common of them being *supervised learning*. In this method of learning, a set of inputs is provided to the NN and its output is compared with the desired output. The difference of actual and desired outputs is used to adjust the connection weights to different elements in the network. (The weights are commonly set randomly at the beginning of the learning process). The process of adjusting weights is repeated until the output falls within an acceptable range. Depending on the application, the training phase may require a lot of computing resources or time. The structure of a NN and the initial conditions are also important parameters for NN training efficiency [Caudill 1990].

Just like other data-processing tools, the age-old principle of "garbage-in, garbage-out" applies to NN's. To ensure a robust NN design, the set of input data and corresponding output data must be chosen carefully. The input-output data set for an NN is called a *training set*.

Additionally, special attention must be paid to the formatting and scaling for the data for effective NN training [Caudill 1990].

The available data is divided into training and *validation sets*. An NN is only trained with the training data. Validation data is *run* on the NN to verify that the inputs are producing desirable outputs. If the validation phase produces large deviations, the training set or the network structure needs to be re-examined; re-training is required in this case. Sometimes, the examination of weights may reveal the reasons for undesirable outputs. Selected test data can be used to make sure that the proper neurons are firing correctly [Caudill 1990].

The learning process is also dependent on the *learning rule*; one such common rule is the *delta rule* that states that if there is a difference between the actual outputs and the desired outputs during training, adjust the weights to reduce the difference. With an input X, if we get Y as the actual output; and Z is the desired output, then we use the following equation to change weights, according to the delta rule [UHouston 2005]:

$$w_{new} = w_{old} + \lambda * (Y - Z) * X \qquad \{12\}$$

where λ is the *learning rate* (e.g., 0.1); w_{old} and w_{new} are the weights before and after the adjustment, respectively.

4.1.4 Motivation

The typical simulation time for each TC, BC, or CPC configuration is between 2-6 hours (on a Windows-XP based, 2.4 GHz Pentium-4 personal computer). This means that hundreds of machine-hours may be spent simulating a reasonable number of cache configurations. In this dissertation, we show that it is feasible to produce NNM's that let a user study the cache performance in seconds (instead of days or weeks) [Caudill 1990].

4.2 Neural Network Modeling for TC, BC, and CPC

4.2.1 Experimental Methodology

We used an NN-modeling software package called Brain Maker (version 3.75) [CalSci 1998] to create and test our NNM's. The software was run on a Windows-XP based, 2.4 GHz Pentium-4 personal computer. Brain Maker's *back-propagation* NN's were 'fully connected', meaning all inputs were connected to all hidden neurons, and all hidden neurons were connected to the outputs. The activation function for hidden and output layers was a sigmoid function. The difference between the network's actual output and the desired output is treated as the error that is to be minimized.

We acquired a total of 150 samples (also called *facts/training facts*) during our simulations for single-threaded versions of TC, BC, and CPC.

120 samples were used as the training set, while the remaining 30 were used as the *validation* set. We stopped an NN training session, when one of these two conditions was met:

(1) Epoch count reached 30000

(2) Ninety percent of the facts were learnt with less than 5% mean squared error

Thirty thousand epochs were used as the training limit because most of the properly converged NN's attained the desired training accuracy before this epoch limit. The use of 90% accuracy allowed us to keep the NN topologies relatively small in size. The 90% threshold also enabled the NN's to achieve generalized instead of 'rote' learning.

4.2.2 Input-Output Definition

The purpose of the NNM's in this research is to predict the values of two parameters for TC, BC, and CPC: trace miss rate and average trace length. These two parameters would be the NNM outputs. Inputs to the NNM's are (1) counts of blocks of different sizes, representing a program (benchmark), (2) cache size, and (3) cache type. Cache type is a symbol, rather than a value, so we used 3 discrete inputs to represent the cache types: TC = {1, 0, 0}; BC = {0, 1, 0}; CPC = {0, 0, 1}. For the purposes of creating NNM's, we kept BBC size and associativity constant so these parameters did not need to be included as NNM inputs.

Regarding the properties of a benchmark as input parameters, our initial attempts involved using a single value for the average block size (for the complete run). But, we discovered that block size averages among the benchmarks were not distinct enough to properly represent the latter for the purposes of NN-training. So, for each benchmark, we used more than one value of block count. (Block sizes and count distribution are shown in Figure 12 on page 27).

Four of the several NNM configurations, we experimented with are shown in Table 9. We selected Configurations 2 and 4 for the final NNM's due to their higher test accuracy.

Table 9. Neural Network Configurations - Input and Output Neurons

Neurons	**Configuration 1**	**Configuration 2**	**Configuration 3**	**Configuration 4**
Output	trace miss rate	trace miss rate	average trace length	average trace length
Input 1	% of blocks with 1 to 4 instructions	% of blocks with 1 to 4 instructions	% of blocks with 1 to 4 instructions	% of blocks with 1 to 4 instructions
Input 2	% of blocks with 5 or more instructions	% of blocks with 5 to 8 instructions	% of blocks with 5 or more instructions	% of blocks with 5 to 8 instructions
Input 3	cache type is TC	% of blocks with 9 to 12 instructions	cache type is TC	% of blocks with 9 to 12 instructions
Input 4	cache type is BC	% of blocks with 13 or more instructions	cache type is BC	% of blocks with 13 or more instructions
Input 5	cache type is CPC	cache type is TC	cache type is CPC	cache type is TC
Input 6	cache size	cache type is BC	cache size	cache type is BC
Input 7		cache type is CPC		cache type is CPC
Input 8		cache size		cache size

4.2.3 Data Pre-Processing

Pre-processing the training and validation sets takes a considerable amount of resources for a practical and reliably functioning NN [Lawrence 1994], [Yale 1997]. In our research, the first data pre-processing step was to apply z-score normalization, a statistical technique of specifying the degree of deviation of a data value from the mean. Stated alternately, z-score places different data on a common scale. Z-score is calculated by this formula [Meas 2005]:

$$Z = \frac{(x - \bar{x})}{\sigma} \qquad \{13\}$$

where $\bar{x}$ is the sample mean, and σ is the sample standard deviation defined as [Triola 1994]:

$$\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{n - 1}} \qquad \{14\}$$

where n is the sample size

As a 2nd step of pre-processing, we normalized the training set to the range [0, 1]; normalization was done to ensure equitable distribution of importance among inputs. In other words, the larger absolute values of an input should not have more influence than the inputs with smaller magnitudes [Masters 1994]. Similarly, we also normalized the outputs to the [0, 1] range [Wolfe & Vemuri 2003]. For n samples, the [0, 1] normalization was a 2-step process:

$$x_i^{'} = x_i - x_{\min}, \quad i = 0..n-1 \qquad \{15\}$$

$$x_i^{''} = x_i^{'} / x_{\max}^{'}, \quad i = 0..n-1 \qquad \{16\}$$

For the cache-size input values that are multiplicative in nature (i.e., 1K, 2K, 4K …), we used log_2 transformation prior to normalization [Masters 1994]. One should, in order to 'use' or 'run' a trained NN, de-normalize (and de-transform, if needed) the predicted outputs to the original ranges.

4.2.4 Neural Network Training and Testing

To find the optimum topologies for our NNM's, we experimented with up to 3 hidden layers; each layer consisted of a different number of neurons. A general rule is that an increase in a number of hidden layers increases as the prediction performance goes up to a certain point, after which the NNM performance starts to deteriorate [Caudill 1990]. The details of some of our NNM's experiments are listed in Table 10 (trace miss rate) and

Table 11 (average trace length). The performance metric for an NNM was the percentage of "training facts" learned with <5% accuracy. For Configuration-2, the training accuracy we were able to achieve was 91%; whereas, Configuration-4 was only able to train with 82% accuracy. Similarity in the values of block-size parameters in the training set seems

to be the reason for difficulty in training the Configuration-4 NN with higher accuracy.

Table 10. Training performance for trace miss-rate NNM ("Configuration-2"): optimum results were achieved with a 4-layer (6-5-5-1) NNM (shown in bold)*

NN size (neurons)	Input layer	6	6	6	6	**6**	6	6	6
	Hidden layer 1	10	20	10	7	**5**	10	15	10
	Hidden layer 2			5	5	**5**	10	10	10
	Hidden layer 3								5
	Output layer	1	1	1	1	**1**	1	1	1
Stop training when	Epoch	30000	30000	14500	30000	**16471**	2900	5167	3008
	90% good facts	no	no	yes	no	**yes**	yes	yes	Yes
Training accuracy		71%	69%	91%	71%	**91%**	91%	91%	91%

* Brain Maker training parameters: Training tolerance = 0.1; testing tolerance = 0.1; learning rate (initial value) = 0.1; learning rate adjustment type = exponential (Refer to [CalSci 1998] for details)

Table 11. Training performance for trace-length NNM ("Configuration-4"): optimum results were achieved with a 4-layer (6-15-10-1) NNM (shown in bold)

NN size (neurons)	Input layer	6	6	6	6	6	6	**6**	6
	Hidden layer 1	10	20	10	7	5	10	**15**	10
	Hidden layer 2			5	5	5	10	**10**	10
	Hidden layer 3								5
	Output layer	1	1	1	1	1	1	**1**	1
Stop training when	Epoch	30000	30000	30000	30000	30000	30000	**30000**	30000
	90% good facts	no	no	no	no	no	no	**no**	no
Training accuracy		78%	35%	68%	37%	37%	76%	**82%**	62%

* Brain Maker training parameters: Training tolerance = 0.1; testing tolerance = 0.1; learning rate (initial value) = 0.1; learning rate adjustment type = exponential (Refer to [CalSci 1998] for details)

4.2.5 Experimental Results and Analysis

Due to the inherent nature of an NNM approach, the input values used for running an NNM should be kept somewhat close to, but not necessarily the same as, the input values in the training set. Significant deviations of the running set from the training set can provide misleading results [Caudill 1990]. We used an arbitrary set of values for block sizes {0.80, 0.17, 0.03, 0.02} (that was different from any of the benchmarks' block sizes), and used it to predict the miss rate and trace length for

different sizes of TC, BC, and CPC. Time to run the NNM's for the above set of inputs was less than a second.

The predicted values of the miss rate are shown in Figure 57. These values resemble the miss rates observed in actual simulations. For the block sizes used with these NN runs, we see that miss rates improve as cache size increases, but the improvement tends to flatten out after 8K cache size. CPC offers better miss rates than TC and BC for all cache sizes.

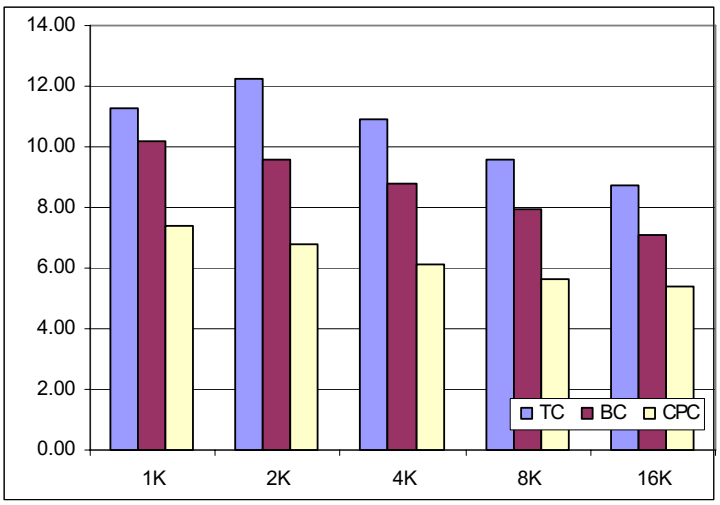

Figure 57. For a program with arbitrarily chosen 'block size distribution' {0.80, 0.17, 0.03, 0.02}, miss-rate NNM was used to predict the values for TC, BC, and CPC. The horizontal axis shows cache size in KB and the vertical axis represents miss rate percentages.

NNM predictions for trace lengths, when cache sizes vary, are shown in Figure 58. Trace lengths for a given cache scheme remain relatively stable, while CPC maintains its lead over both TC and BC.

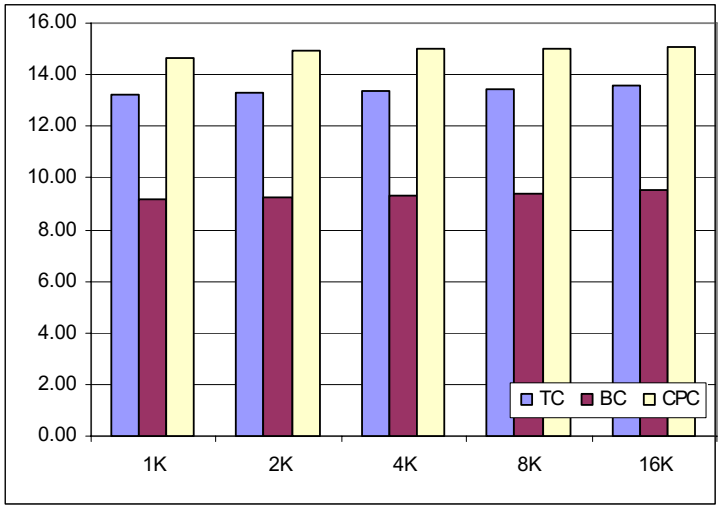

Figure 58. For a program with arbitrary chosen 'block size distribution' {0.80, 0.17, 0.03, 0.02}, trace-length NNM was used to predict the values for TC, BC, and CPC. The horizontal axis shows cache size in KB and the vertical axis represents the trace length in terms of number of instructions.

4.3 Conclusions

The results from NNM experiments demonstrate that the NNM are capable of learning the trace-based caches' input-output mapping functions, in the encoded form of the weights of the neurons. We can also say that an NNM can be used as a time-efficient alternative to simulations. Further research and investigation into trace-based cache NNM's can be conducted using more input parameters, as well as a

larger training set. Branch types and frequencies for benchmarks (or programs) can be used as additional inputs.

CHAPTER V

CONCLUSIONS & FUTURE RESEARCH

In Section 5.1 of this last chapter, we summarize our findings, and in Section 5.2, we present our research contributions. Then, we conclude by proposing a number of ideas for extending this research.

With the ongoing availability of a larger number of faster transistors to a designer, it may be a challenge to make judicious use of these newer devices, primarily because of power consumption and clock speed limitations. In this context, we have presented a better scheme for the utilization of resources by introducing a new instruction cache architecture. The cache is used with superscalar processors and is called code pattern cache (CPC). CPC operates on the basic principles that common programs tend to exhibit repeatability in their execution patterns, and that making efficient use of captured dynamic instruction sequences can enhance the performance of the instruction fetch mechanisms.

5.1 Conclusions

In this dissertation, we demonstrated that a larger instruction fetch bandwidth could be achieved with less complexity and smaller cache capacity. In short, the following techniques were used to attain higher performance: (1) removal of redundancy of instruction storage, inherent to trace cache; (2) accommodation of basic blocks of varying lengths (vs. the fixed length blocks of block cache); (3) reduction of die area and resulting power consumption by storing basic blocks in only one place, instead of multiple locations, as in block cache; (4) inclusion of (traditional) way-associativity; and (5) accessing basic block data pointers and data arrays simultaneously to reduce latency. In the following paragraphs, we list the conclusions:

- We have presented CPC, a new instruction cache architecture for improving instruction fetch rates beyond current trace-based cache schemes (TC and BC). CPC stores basic blocks that are not fixed in size in the BBC structure. The boundary addresses that are used to form traces are stored in a separate set-associative structure called BPC. Both BPC and BBC are looked up concurrently to determine a trace hit. In CPC, basic block overlapping that is inherent to TC has been eliminated and CPC does not need the redundant cache storage (and related hardware

complexity) that BC does. Multi-threading capabilities have been incorporated into CPC.

- We developed functional simulators for three cache schemes: TC, BC and CPC. The simulators were developed in VHDL and were used to run single-threaded SPECint2000 benchmark programs. Simulators were flexible enough to accommodate variability in cache size, block size and count or trace size, way-associativity, etc. The simulators did not include any instruction execution capability and functioned by reading in program traces pre-saved from a simulator such as sim-cache [Burger & Austin 1997]. The simulators produced two statistics at the end of a simulation: trace miss rate and average trace length. From the simulation results, we showed that CPC had better trace miss rates and longer average traces than both TC and BC. For 10 SPECint2000 benchmarks we used in this research, CPC's average miss rate was better than TC by 73.7%, and was 22.7% better than BC. On average, CPC's traces were 79.7% longer than TC's traces, and 106.1% longer than BC's traces.
- The multi-threaded versions of our TC, BC, and CPC simulators allowed instantiation of any number of threads, as long as the simulation platform performed simulations in a reasonable amount

of time. Our research utilized 2, 4, 8, and 16-threaded versions of all three caches. The simulation times of multi-threaded caches were significantly longer than their single-thread counterparts. We created 10 different multi-threaded workloads by running SPECint2000 benchmarks on multiple threads. When measured in terms of trace miss rate and average trace length, CPC sustained its performance lead over TC and BC in multi-thread configurations. CPC, generally, exhibited higher performance gain over other caches while multi-threading than it did while single-threading. CPC's miss rate reduction as compared to TC was 85.7%, and 36% as compared to BC. CPC's traces were 86.1% longer than TC's traces, and 98.4% longer than BC's.

- We used CACTI, a readily available modeling tool, to compare the power, area, and access times of TC, BC, and CPC. BC was found to be behind TC and BC in terms of three modeled parameters; the main reason for the lag was the (4x) replication of block cache structures in BC. CPC had higher power consumption and took up more die area than TC; CPC had comparable access time with TC.
- In order to perform a mutually consistent comparison of TC, BC, and CPC, a new metric called aggregate performance index (API) was introduced. The metric combined the simulation and modeling

results. Although CPC had higher power consumption and more die area than TC, API showed that CPC still had better overall performance than both TC and BC.

- We created unified NNM's for TC, BC, and CPC to demonstrate an NNM's feasibility as an alternative to cache simulations (As of the time of writing of this dissertation, no other such models for caches had been reported in the research publications). One NNM was used to predict the trace miss rate for all three caches, and the other to predict the average trace length. Training accuracy of the miss-rate-NNM was 91%; trace-length-NNM was only able to train with 82% accuracy. With these accuracies, the NNM's provided a good estimation of non-linear behavior of TC, BC, and CPC, without delving into the details of the caches' internal working. The NNM's seemed to be a viable substitute to the otherwise very time-consuming simulations. The NNM's produced modeling results in a fraction of a second, as compared to 4 to 6-hour simulation time of a given cache. For a program represented by an arbitrary set of input values (of block size distribution and cache size), we used the NNM's to predict the cache performance. The trends in cache performance predicted by the NNM's resembled the simulation results.

5.2 Future Research

Below we have listed some areas in which the research on CPC can be extended:

5.2.1 CPC Architecture & Simulations

- Our research employed simulations involving only a single-cache hierarchy. In the future, a full-processor model could be developed to study other aspects of CPC performance such as IPC.
- More advanced branch prediction schemes could be used to further enhance CPC's miss rate performance.
- Characterization of branch types and taken/not-taken frequencies in the benchmarks could help further analyze the simulation results for miss rates and trace lengths.
- Multi-threading studies could be further expanded to include the effect of thread-shared BPC.
- To better understand CPC's performance in multi-threaded environment, an in-depth study of the cross-thread clobbering effect (in thread-shared BPC and/or BBC) could be performed.

5.2.2 Power, Area, and Access Time Modeling

- Power, area, and timing models could be made more accurate with the inclusion of auxiliary logic required for cache operation, e.g., CPC's merging buffer.

- Effect of sub-banking, read-write port-counts on the power, area, and timing aspects of a cache could be studied.
- Access time and miss rates for different caches could be used to calculate consumed energy.

5.2.3 Neural Network Modeling

- The training set for NNM could be expanded to include a larger set of input parameters, for example, cache-associativity, trace capacity, thread count, etc.
- Simulation results from benchmarks, other than the 10 that we used in this research, could be included in the training set.

APPENDIX

SPECINT2000 BENCHMARKS

We used ten programs from SPEC 2000 integer benchmarks suite in our simulations of TC, BC, and CPC. The programs were listed in Table 2 (page 66) and are described here [Spec 2000].

256.bzip2

Type: Compression

Description:

"256.bzip2 is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and 256.bzip2 is that SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory which helps isolate the work done to only the CPU and memory subsystem."

186.crafty

Type: Game playing program (plays chess)

Description:

"Crafty is a high-performance Computer Chess program that is designed around a 64-bit word. It runs on 32 bit machines using the "long long" (or similar, as _int64 in Microsoft C) data type. It is primarily an integer code, with a significant number of logical operations such as and, or, exclusive or and shift. It can be configured to run a reproducible set of searches to compare the integer/branch prediction/pipe-lining facilities of a processor."

254.gap

Type: Group theory, interpreter

Description:

"It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming)."

176.gcc

Type: C Language optimizing compiler

Description:

"176.gcc is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled. 176.gcc has had its inlining heuristics altered slightly, so as to inline more code than would be typical on a UNIX system in 1997. It is expected that this effect will be more typical of compiler usage in 2002 which was done so that 176.gcc would spend more time analyzing it's source code inputs, and use more memory. Without this effect, 176.gcc would have done less analysis, and needed more input workloads to achieve the run times required for SPECint2000."

181.mcf

Type: Combinatorial optimization / Single-depot vehicle scheduling

Description:

"A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C; the benchmark version uses almost exclusively integer arithmetic."

"The program is designed for the solution of single-depot vehicle scheduling (sub-) problems occurring in the planning process of public transportation companies. It considers one single depot and a homogeneous vehicle fleet. Based on a line plan and service frequencies, so-called timetabled trips with fixed departure/arrival locations and times are derived. Each of these timetabled trips has to be serviced by exactly one vehicle. The links between these trips are so-called dead-head trips. In addition, there are pull-out and pull-in trips for leaving and entering the depot."

"Cost coefficients are given for all dead-head, pull-out, and pull-in trips. It is the task to schedule all timetabled trips to so-called blocks such that the number of necessary vehicles is as small as possible and, subordinate, the operational costs among all minimal fleet solutions are minimized."

"For simplification in the benchmark test, we assume that each pull-out and pull-in trip is defined implicitly with duration of 15 minutes and a cost coefficient of 15."

"For the considered single-depot case, the problem can be formulated as a large-scale minimum-cost flow problem that we solve with a network simplex algorithm accelerated with a column generation. The core of the benchmark 181.mcf is the network simplex code "MCF Version 1.2 - A network simplex implementation". For this benchmark, MCF is embedded in the column generation process."

"The network simplex algorithm is a specialized version of the well known simplex algorithm for network flow problems. The linear algebra of the general algorithm is replaced by simple network operations such as finding cycles or modifying spanning trees that can be performed very quickly. The main work of our network simplex implementation is pointer and integer arithmetic."

197.parser

Type: Word processing

Description:

"The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words."

"The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare

and idiomatic ones. The parser is robust; it is able to skip over portions of the sentence that it cannot understand, and assign some structure to the rest of the sentence. It is able to handle unknown vocabulary, and make intelligent guesses from context about the syntactic categories of unknown words."

253.perlbmk

Type: Programming language

Description:

"253.perlbmk is a cut-down version of Perl v5.005_03, the popular scripting language. SPEC's version of Perl has had most of OS-specific features removed. In addition to the core Perl interpreter, several third-party modules are used: MD5 v1.7, MHonArc v2.3.3, IO-stringy v1.205, MailTools v1.11, TimeDate v1.08."

255.vortex

Type: Database

Description:

"VORTEx is a single-user object-oriented database transaction benchmark which exercises a system kernel coded in integer C. The VORTEx benchmark is a derivative of a full OODBMS that has been customized to conform to SPEC CINT2000 (component measurement) guidelines."

"The benchmark 255.vortex is a subset of a full object oriented database program called VORTEx. (VORTEx stands for "Virtual Object Runtime EXpository.")"

"Transactions to and from the database are translated though a schema. (A schema provides the necessary information to generate the mapping of the internally stored data block to a model viewable in the context of the application.)"

175.vpr

Type: Integrated Circuit Computer-Aided Design Program (More specifically, performs placement and routing in Field-Programmable Gate Arrays)

Description:

"VPR is a placement and routing program; it automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs."

"Placement consists of determining which logic block and which I/O pad within the FPGA should implement each of the functions

required by the circuit. The goal is to place pieces of logic which are connected (i.e. must communicate) close together in order to minimize the amount of wiring required and to maximize the circuit speed. This is basically a slot assignment problem - assign every logic block function required by the circuit and every I/O function required by the circuit to a logic block or I/O pad in the FPGA, such that speed and wire-minimization goals are met. VPR uses simulated annealing to place the circuit. An initial random placement is repeatedly modified through local perturbations in order to increase the quality of the placement, in a method similar to the way metals are slowly cooled to produce strong objects."

"Routing (in an FPGA) consists of determining which programmable switches should be turned on in order to connect the pre-fabricated wires in the FPGA to the logic block inputs and outputs, and to other wires, such that all the connections required by the circuit are completed and such that the circuit speed is maximized. The connections required by the circuit are represented as a hypergraph, and the possible connections of wire segments to other wires and to logic block inputs and outputs are represented by (a different) directed graph, which is often called a "routing-resource" graph."

"VPR uses a variation of Dijkstra's algorithm in its innermost routing loop in order to connect the terminals of a net (signal) together. Congestion detection and avoidance features run "on top" of this innermost algorithm to resolve contention between different circuit signals over the limited interconnect resources in the FPGA."

REFERENCES

[1] [Amrutur & Horowitz 2000] Amrutur, B. S., and M. A. Horowitz, "Speed and power scaling of SRAM's," *IEEE Journal of Solid-State Circuits*, Vol. 35, Issue 2, Feb. 2000, pp. 175-85.

[2] [Banakar et al. 2002] Banakar, R., S. Steinke, L. Bo-Sik, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, May 2002, pp. 73-78.

[3] [Batson & Vijaykumar 2001] Batson, B., and T.N. Vijaykumar, "Reactive-associative caches," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2001, pp. 49–60.

[4] [Bigus 1994] Bigus, J.P., "Applying neural networks to computer system performance tuning," *Proceedings of IEEE International Conference on Computational Intelligence, Neural Networks*, Vol. 4, Jul. 1994.

[5] [Black et al. 1999] Black, B., B. Rychlick, and J. Shen, "The block-based trace cache," *Proceedings of the 26th International Symposium on Computer Architecture*, 1999, pp. 196–207.

[6] [Burger & Austin 1997] Burger, D., and T. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1242, Jun. 1997.

[7] [CalSci 1998] "BrainMaker – User's Guide and Reference Manual," 7th ed., California Scientific Software Press, Jun. 1998.

[8] [Caudill 1990] Caudill, M., "AI Expert: Neural Network Primer," Miller Freeman Publications, 1990.

[9] [Conte et al. 1995] Conte, T., K. Menezes, P. Mills, and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates," *22nd International Symposium on Computer Architecture*, Jun. 1995, pp. 333-344.

[10] [Dutta & Franklin 1995] Dutta, S., and M. Franklin, "Control flow prediction with tree-like sub-graphs for superscalar processors," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 258–263.

[11] [Dutta & Franklin 1999] Dutta, S., and M. Franklin, "Control flow prediction schemes for wide-issue superscalar processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, Issue 4, Apr. 1999, pp. 346–359.

[12] [El-Moursy & Albonesi 2003] El-Moursy, A., and D. H. Albonesi, "Front-end policies for improved issue efficiency in SMT processors," *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, Feb. 2003.

[13] [Govindarajan et al. 1995] Govindarajan, R., S. S. Nemawarkar, and P. LeNir, "Design and performance evaluation of a multithreaded architecture," *Proceedings of First IEEE Symposium on High-Performance Computer Architecture*, Jan. 1995.

[14] [Gruen & Kubota 2002] Gruen, R., and T. Kubota, "A neural network approach to system performance analysis," *Proceedings of IEEE SoutheastCon*, Apr. 2002.

[15] [Gummaraju & Franklin 2000] Gummaraju, J., and M. Franklin, "Branch prediction in multi-threaded processors," *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2000.

[16] [Hanson et al. 2003] Hanson, H., M. S. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger, "Static energy reduction techniques for microprocessor caches," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 11, Issue 3, Jun. 2003, pp. 303–313.

[17] [Hao et al. 1996] Hao, E., P. Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured Instruction Set Architectures," *Proceedings of the 29th Annual*

IEEE/ACM International Symposium on Microarchitecture, pp. 191–200, 1996.

[18] [Harper et al. 1999] Harper, J.S., D.J. Kerbyson, and G.R. Nudd, "Analytical modeling of set-associative cache behavior," *IEEE Transactions on Computers*, Vol. 48, Issue 10, Oct. 1999.

[19] [Hennessy & Patterson 2003] Hennessy, J., and D. Patterson, "Computer Architecture: A Quantitative Approach," 3rd ed., Morgan Kaufman Publishers, Inc, CA, 2003.

[20] [Hily & Seznec 1996] Hily, S., and A. Seznec, "Branch prediction and simultaneous multithreading", *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, Oct. 1996, pp.169–173.

[21] [Hossain 2002] Hossain, A., "Trace Cache in Simultaneous Multi-threading," PhD dissertation, Dept. of Computer Engineering, Syracuse University, 2002.

[22] [Hossain et al. 2002] Hossain, A., D. J. Pease, J. S. Burns, and N. Parveen, "Trace cache performance parameters," *Proceedings of IEEE International Conference on Computer Design, VLSI in Computers and Processors*, Sep. 2002.

[23] [Howard & Lipasti 1999] Howard, D. L., and M. H. Lipasti, "The effect of program optimization on trace cache efficiency," *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 1999, pp. 256–261.

[24] [Intel 1997] "Pentium II Processor Developer's Manual," Intel Corporation, 1997.

[25] [Intel 2001] "Multi-Threaded Programming for Next Generation Multi-Processing Technology," Intel Corporation, Aug. 2001. [Online]. Available: http://www.intel.com/technology/hyperthread

[26] [Jacob & Mudge 1998] Jacob, B. and T. Mudge, "Virtual memory: issues of implementation", *Computer*, Vol. 31, Issue 6, Jun. 1998, pp. 33–43.

[27] [Jimenez & Lin 2001] Jimenez, D. A., and C. Lin, "Dynamic branch prediction with perceptrons," *The Seventh International*

Symposium on High-Performance Computer Architecture, Jan. 2001, pp. 197–206.

[28] [Jourdan et al. 2000] Jourdan, S., L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, "Extended block cache," *Proceedings of Sixth International Symposium on High-Performance Computer Architecture*, Jan. 2000.

[29] [Kavi et al. 1995] Kavi, K. M., A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of cache memories for multi-threaded dataflow architecture," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Jun. 1995.

[30] [Khalid & Obaidat 2000] Khalid, H., and M. S. Obaidat, "KORA: a new cache replacement scheme," *Computers & Electrical Engineering,* Vol. 26, Issues 3-4, Apr. 2000, pp. 187-206.

[31] [Khalid 1996] Khalid, H., "A neural network-based replacement strategy for high performance computer architectures," PhD dissertation, Department of Electrical Engineering, City University of New York, The City College, New York, USA, Jun. 1996.

[32] [Lawrence 1994] Lawrence, J., "Introduction to Neural Networks – Design, Theory and Applications," California Scientific Software Press, 1994.

[33] [Lioupis & Milios 1997] Lioupis, D., and S. Milios, "The effects of cache architecture on the performance of operating systems in multi-threaded processors," *Proceedings of Ninth Euromicro Workshop on Real-Time Systems*, Jun. 1997, pp. 72-79

[34] [Marr 2002] Marr, D., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, Q1 2002. [Online]. Available: http://www.intel.com/technology/hyperthread/

[35] [Masters 1994] Masters, T., "Signal and Image Processing with Neural Networks," John Wiley & Sons, Inc., 1994.

[36] [McBader & Lee 2003] McBader, S., and P. Lee, "Reducing memory bottlenecks in embedded, parallel image processors," *Electronics Letters*, Vol. 39, Issue 1, Jan. 2003.

[37] [McFarling 1993] McFarling, S., "Combining branch predictors," Tech. Report TN-36, Digital Western Lab, Jun. 1993.

[38] [Meas 2005] [Online]. Available: http://www.measuringusability.com/z.htm

[39] [Moore 1965] Moore, G. E., "Cramming more components onto integrated circuits", *Electronics*, Vol. 38, No. 8, Apr. 1965.

[40] [Noonburg & Shen 1994] Noonburg, D.B., and J.P. Shen, "Theoretical modeling of superscalar processor performance," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO-27, 30 Nov.-2 Dec. 1994, pp. 52–62.

[41] [Ozturk et al. 2005] Ozturk, O., M. Kandemir, M.; M. J. Irwin, "BB-GC: basic block level garbage collection", *Proceedings of Design, Automation and Test in Europe*, March 2005, pp. 1032–1037.

[42] [Patel et al. 1998] Patel, S. J., M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," *Proceedings of 25th Annual International Symposium on Computer Architecture*, 1998, pp. 262–271.

[43] [Patel et al. 1999] Patel, S., D. Friendly, and Y. N. Patt, "Evaluation of design option for the trace cache fetch mechanism," *IEEE Transactions on Computers*, Vol. 48, Issue 2, Feb. 1999, pp. 193–204.

[44] [Ramirez et al. 2000] Ramirez, A., J. Larriba-Pey, and M. Valero, "Trace cache redundancy: red and blue traces," *Proceedings of Sixth International Symposium on High-Performance Computer Architecture*, Jan. 2000, pp. 325–333.

[45] [Rotenberg et al. 1999] Rotenberg, E., S. Bennett, and J. E. Smith, "A trace cache microarchitecture and evaluation," *IEEE Transactions on Computers*, Vol. 48, Issue 2, Feb. 1999, pp. 111–120.

[46] [Sangireddy et al. 2004] Sangireddy, R., H. Kim, and A. K. Somani, "Low-power high-performance reconfigurable computing cache architectures," *IEEE Transactions on Computers*, Vol. 53, Issue 10, Oct. 2004, pp. 1274-1290.

[47] [Schlansker et al. 1997] Schlansker, M., T.M. Conte, J. Dehnert, K. Ebcioglu, J. Z. Fang, and C. L. Thompson, "Compilers for instruction-level parallelism," *Computer*, Vol. 30, Issue 12, Dec. 1997, pp. 63–69.

[48] [Shanley & Anderson 1995] Shanley, T., and D. Anderson, "ISA System Architecture," Addison-Wesley Publishing Company, 1995.

[49] [Shivakumar & Jouppi 2001] Shivakumar, P., and N. P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," Technical Report, WRL Research Report 2001/2, Compaq Western Research Laboratory, Aug. 2001.

[50] [Simple 2005] [Online]. Available: http://www.simplescalar.com

[51] [Simpson et al. 1997] Simpson, T. W., J. Peplinski, P. N. Koch, and J. K. Allen, "On the use of statistics in design and the implications for deterministic computer experiments," *Proceedings of ASME Design Engineernig Technical Conferences*, Sep. 14-17, 1997.

[52] [Singh et al. 1992] Singh, J. P., H. S. Stone, and D. F. Thiebaut, "A model of workloads and its use in miss-rate prediction for fully associative caches," *IEEE Transactions on Computers*, Vol. 41, Issue 7, Jul. 1992.

[53] [Smith 1994] Smith, A. J., "The need for measured data in computer system performance system analysis," *Proceedings of Eighteenth Annual International Computer Software and Applications Conference*, Nov. 1994.

[54] [Sobecks et al. 1998] Sobecks, B., J. Nevin, and A. Helmicki, "Performance modeling of analog circuits via neural networks: the design process view," *Proceedings of Midwest Symposium on Circuits and Systems*, Aug. 1998.

[55] [Spec 2000] [Online]. Available: http://www.spec.org/cpu2000/CINT2000/

[56] [Stegmayer & Chiotti 2004] Stegmayer, G., and O. Chiotti, "The Volterra representation of an electronic device using the Netural Network parameters," *Latin American Conference on Informatics (CLEI'2004)*, Sep. 2004.

[57] [Triola 1994] Triola, M., "Elementary Stastictics," 6th ed., Addison-Wesley Publishing Co, 1994.

[58] [Tullsen et al. 1995] Tullsen, D. M., S. J. Eggers, and H. M. Levy, "Simultaneous multi-threading: maximizing on-chip parallelism," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Jun. 1995.

[59] [UHouston 2005] [Online]. Available: http://www2.cs.uh.edu/~vilalta/courses/machinelearning/neuralnetworks1.ppt

[60] [Uhrig 1995] Uhrig, R. E., "Introduction to artificial neural networks," *Proceedings of the 1995 IEEE IECON 21st International Conference on Industrial Electronics, Control and Instrumentation*, Vol. 1, Nov. 1995.

[61] [UTexas 2005] [Online]. Available: http://www.eco.utexas.edu/faculty/Kendrick/frontpg/NeuralNets.htm

[62] [Wada et al. 1992] Wada, T., S. Rajan, and S. A. Przybylski, "An analytical access time model for on-chip cache memories," *IEEE Journal of Solid-State Circuits*, Vol. 27 Issue 8, Aug. 1992, pp. 1147-1156.

[63] [Wallace & Bagherzadeh 1998] Wallace, S., and N. Bagherzadeh, "Modeled and measured instruction fetching performance for superscalar microprocessors," *IEEE Transaction on Parallel and Distrubuted Systems*, Vol. 9, No. 6, Jun. 1998.

[64] [Wilton & Jouppi 1996] Wilton, S. J., and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, Vol. 31, Issue 5, May 1996, pp. 677–688.

[65] [Wolfe & Vemuri 2003] Wolfe, G., and R. Vemuri, "Extraction and use of neural network models in automated synthesis of operational amplifiers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22, Issue 2, Feb. 2003.

[66] [Yale 1997] Yale, K., "Preparing the right data for training neural networks," *IEEE Spectrum*, Vol. 34, Issue 3, Mar. 1997, pp. 64-66.

[67] [Yeh & Patt 1992] Yeh, T., and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," *Proceedings of ISCA*, 1992, pp. 124-134.