Cache and Branch Prediction Improvements for Advanced Computer Architecture

by

YUL CHU

B.Sc., KwangWoon University, Korea, 1984.
M.S.E.E., Washington State University, USA, 1995.

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Department of Electrical and Computer Engineering)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

May 2001

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical and Computer Engineering

The University of British Columbia
Vancouver, Canada

Date May 30, 2001

DE-6 (2/88)

ABSTRACT

As the gap between memory and processor performance continues to grow, more and more programs will be limited in performance: by the memory latency of the system and by the branch instructions (control flow of the programs). Meanwhile, due to the increase in complexity of application programs over the last decade, object-oriented languages are replacing traditional languages because of convenient code reusability and maintainability. However, it has also been observed that the run-time performance of object-oriented programs can be improved by reducing the impact caused by the memory latency, branch misprediction, and several other factors. In this thesis, two new schemes are introduced for reducing the memory latency and branch mispredictions for High Performance Computing (HPC).

For the first scheme, in order to reduce the memory latency, this thesis presents a new cache scheme called TAC (Thrashing-Avoidance Cache), which can effectively reduce instruction cache misses caused by procedure call/returns. The TAC scheme employs N-way banks and XOR mapping functions. The main function of the TAC is to place a group of instructions separated by a call instruction into a bank according to the initial and final bank selection mechanisms. After the initial bank selection mechanism selects a bank on an instruction cache miss, the final bank selection mechanism will determine the final bank for updating a cache line as a correction mechanism. These two mechanisms can guarantee that recent groups of instructions exist in each bank safely. A simulation program, TACSim, has been developed by using Shade and Spixtools, provided by SUN Microsystems, on an ultra SPARC/10 processor. Our experimental results show that TAC schemes reduce conflict misses more effectively than skewed-associative caches in both C (9.29% improvement) and C++ (44.44% improvement)

programs on L1 caches. In addition, TAC schemes also allow for a significant miss reduction on Branch Target Buffers (BTB).

For the second scheme to reduce branch mispredictions, this thesis also presents a new hybrid branch predictor called the GoStay2 that can effectively reduce misprediction rates for indirect branches. The GoStay2 has two different mechanisms compared to other 2-stage hybrid predictors that use a Branch Target Buffer (BTB) as the first stage predictor: First, to reduce conflict misses in the first stage, an effective 2-way cache scheme is used instead of a 4-way set-associative scheme. Second, to reduce mispredictions caused by an inefficient predict and update rule, a new selection mechanism and update rule are proposed. A simulation program, GoS-Sim, has been developed by using Shade and Spixtools, provided by SUN Microsystems, on an Ultra SPARC/10 processor. Our results show significant improvement with these mechanisms compared to other hybrid predictors. For example, the GoStay2 improves indirect misprediction rates of a 64-entry to 4K-entry BTB (with a 512- or 1K-entry PHT) by 14.9% to 21.53% compared to the Cascaded predictor (with leaky filter).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I sincerely wish to express my gratitude to my supervisor, Dr. M. R. Ito, for his inspiring and encouraging guidance in leading me to a deeper understanding of this work. His invaluable oral and written comments were always extremely perceptive, helpful, and appropriate.

I would also like to thank my committee members, Dr. Steve Wilton and Dr. Alan Hu, for their help, advice, and valuable suggestions throughout my work. Thanks to Dr. Hu, I was able to use SPEC95 benchmark program for my simulation without any trouble. Thanks to Dr. Wilton, I was able to expand one of my future research programs to include the field of FPGA.

Special thanks should go to my friends Robert Ross and Susan Ritchie who provided valuable proofreading comments on this work.

•

I am also very grateful to Dr. Norm Hutchinson, Dr. Babak Hamidzadeh and Dr. Mark R. Greenstreet for their kindness in participating in my examining committee. I am also very grateful to Dr. Nikitas Dimopoulos for his valuable editorial comments and for passing me as an external examiner.

It is my great pleasure to dedicate this small achievement to my mother Sunghye Hwang, my father Kwangho Chu, my wife Myunghee Chu, our children, Sangjun, Jinna and Yuna for their love, support, and patience over the past several years.

I want to express my deep gratitude to my brother Dr. Hong Chu, my cousin Ronald Choo, and their families for their love and encouragement.

Last, but certainly not least, I would like to extend my deepest thanks to Kyungwoo club members in Korea and many friends for their support and understanding throughout my time at UBC.

This dissertation could not have been completed without the support of the many people who are gratefully acknowledged here.

This research has been funded by the Natural Sciences and Engineering Research Council of Canada.

Chapter 1 Overview and summary

1.1 Introduction

Through the mid-1980s, supercomputers such as Cray were used to achieve high performance for advanced scientific and engineering applications. However, since the late 1980s, supercomputers have not been able to significantly improve performance. They have been restricted by high cost (about $3 million) compared to a personal computer (about $3000) and limited by the number of customers [Dowd & Severance '98].

Meanwhile, the performance of microprocessor architectures has doubled every two to three years. This has occurred for two reasons. First, microprocessor architectures are borrowing and innovating with techniques formerly unique to supercomputers and large mainframes. The second reason has been the emergence of a personal and business computer market which demands high performance for computer usage such as 3D graphics, graphical user interface, and games [Dowd & Severance '98]. However, supercomputers are still used for the most demanding applications such as weather forecasting.

What is High Performance Computing (HPC)?

In general, High Performance Computing (HPC) refers to computing systems that are used to provide solutions to problems that require the significant computational power

needed to process very large amounts of data quickly, and are also needed to operate interactively across a geographically-distributed network. Figure 1 shows an overview of HPC with respect to three different areas: goals, architectures, and techniques.

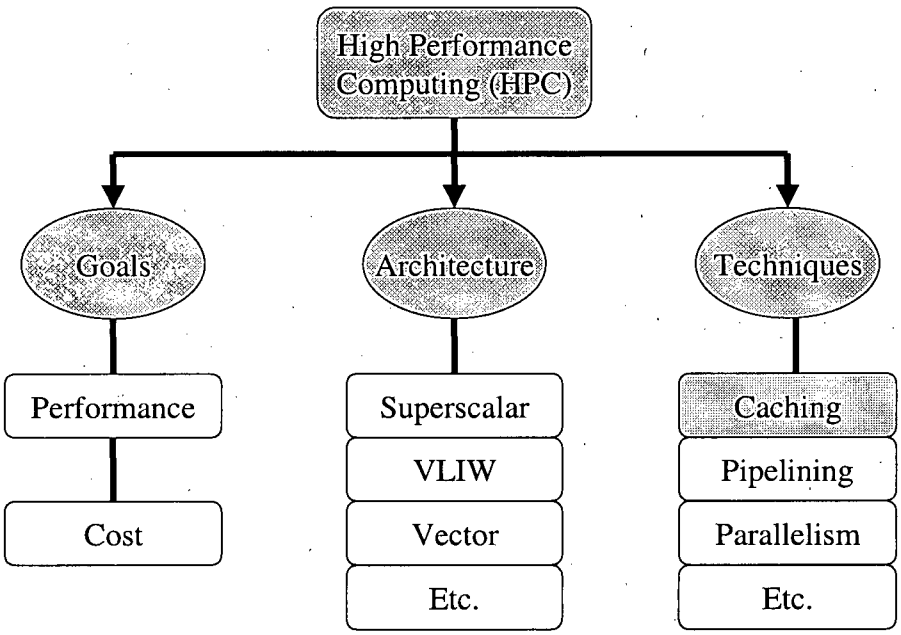


Figure 1. An overview of High Performance Computing (HPC)

In Figure 1, the goals for HPC could be achieved through maximum performance and minimum cost. How to maximize performance depends on reducing the time to execute a program (T), which is a function of the number of instructions to execute (n_i), the average number of clock cycles per instruction (CPI), and the clock cycle time (t_c). From (1), there are two distinct approaches for increasing system performance:

$$T = n_i \times CPI \times t_c \quad \text{----------} \quad (1)$$

- By lowering the clock cycle time (t_c) – Much of this performance gain comes as a consequence of circuit and layout improvement. However, this is becoming increasingly

difficult and will eventually reach physical constraints. Since this topic is beyond the scope of this thesis, it will not be covered in detail.

- By improving two other factors (*CPI, n_i*) – The major performance optimization is pipelining, in which a stream of instructions progress from pipeline stage to pipeline stage with overlapping of instruction fetch, decode, and execution. This technique will be discussed in detail with other techniques such as cache memory, parallelism and superscalar throughout this section.

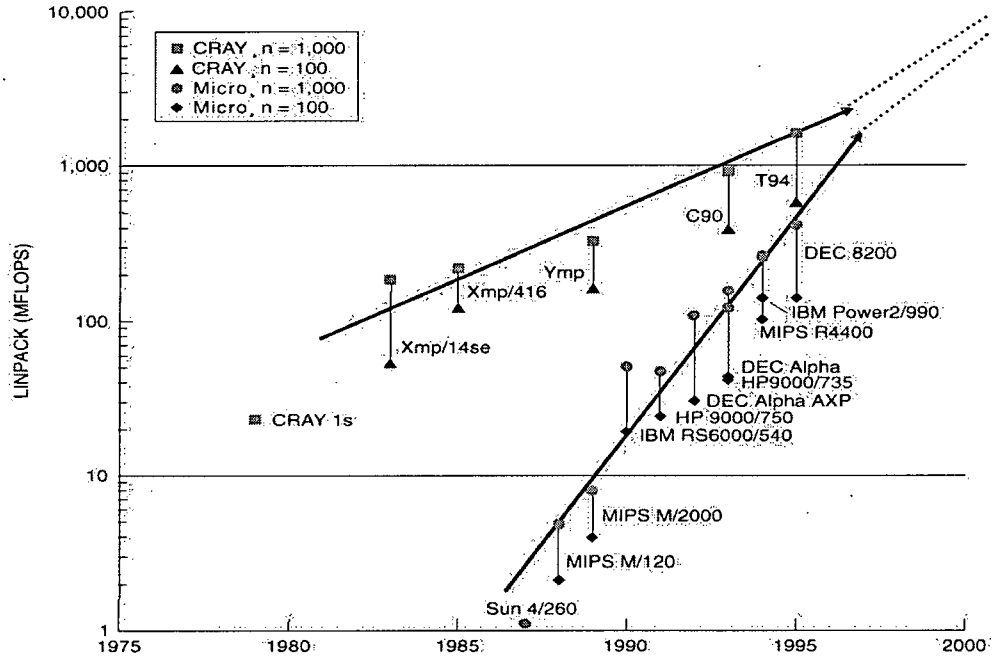


Figure 2. The technical trend for supercomputers and microprocessors [Buyya '00].

Figure 2 compares the performance since 1980 between supercomputers such as CRAY and microcomputers by using the LINPACK benchmark program. The LINPACK is one of the more famous floating-point benchmarks of recent years, created by Jack Dongarra, and gets its name from a linear algebra package that it uses to solve a dense

system of linear equations with Gaussian elimination. The LINPACK keeps track of execution time and then divides this into the number of floating-point operations that are performed to get a MegaFLOPS rating. *'n = 100 and n = 1000'* in Figure 2 describes the results based on a 100 x 100 and 1000 x 1000 matrix using a double-precision floating point. Figure 2 shows that the performance of a supercomputer in 1990 is similar to a microcomputer in 1995. Moreover, the performance gap between supercomputers and microprocessors has been decreasing since 1995 because of the rapid technical development of microprocessors.

In 1988, an article appeared in the Wall Street Journal titled 'Attack of the Killer Micro' that described how computing systems made up of many small ($3,000 to $20,000) machines would soon make large supercomputers ($3 million) obsolete. These inexpensive processors have been developed toward high performance computing systems. HPC, which is broader than supercomputing with supercomputers, is a moving target because of the steady and rapid gains in the performance/cost ratio. Yesterday's supercomputer is today's personal computer; today's leading-edge techniques for supercomputers will be among tomorrow's mainstream capabilities for HPC.

In Figure 1, the architectures for HPC have a tendency to be designed in such a way as to do additional parallelism proportional to increased machine resources [Lipasti & Shen '97]. According to Lipasti & Shen ('97), these architectures are:

- Superscalar machines schedule instructions dynamically at runtime. These machines can reduce the average number of cycles per instruction, but they need extra hardware. Therefore, performance depends on the amount of resources in the machine;

- VLIW machines schedule instructions statically at compile time. These machines contain numerous functional units, which accommodate multiple streams of data input, such as audio and video. In general, VLIW machines heavily rely on powerful compilers to detect and resolve inter-instruction dependencies in software. This keeps the hardware design simple and fast. But their static nature makes them incompatible with dynamic variations in parallelism, which are caused by an aggressive memory subsystem and speculative-execution techniques;
- Multithreaded processors support multiple machine contexts and execute multiple instruction streams simultaneously. The performance depends on finding enough thread parallelism by software. The disadvantages of these machines are that debugging multithread programs is difficult, and that there is a lack of automatic thread-partitioning compilers;
- Single chip multiprocessors are used for improving throughput under multiprogrammed workloads. However, these machines are restricted to numerical applications that contain easily parallelized loops. Limited processor interconnects and synchronization overhead will degrade system performance.
- Vector processors are machines built primarily to handle large scientific and engineering calculations. Their performance derives from a heavily pipelined architecture which operations on vectors and matrices can efficiently exploit. As an example, the NEC SX5/3C is reported at 8 Gflops per second per processor peak.

The techniques for HPC in Figure 1 can be categorized into four different fields. Those fields include:

- *Pipelining* allows increased utilization of hardware resources by the partial execution of more than one instruction at the same time. One of the most common uses of pipelining is to fetch the next instructions from lower level memory while executing a current one.
- *Cache memory* is to improve the throughput of memory data and instruction flow. Memory data flow is relevant to the load/store instructions. The data values are stored and retrieved from data memory. To reduce average memory latency, the prediction of load values and addresses are incorporated into the execution core. Meanwhile, there are two main logical stages in the instruction flow: Fetch – the processor retrieves instructions from cache or main memory; and Decode – the processor decodes instructions, renames their operands, and detects inter-instruction dependencies. For each stage, there is a need to reduce cache misses by using an efficient cache scheme and increase the speculation for the control-flow instructions with an accurate branch predictor.
- For the technique of *parallelism*, multiple execution units are popular for improving performance. The execution core must strive for two fundamental goals to increase instruction throughput. It must:
  - efficiently detect and resolve inter-instruction dependencies; and
  - eliminate or bypass as many dependencies as possible to explore more parallelism between instructions.
- Other techniques such as *prefetching*, *buffering*, etc. are also popular for improving system performance for the HPC.

What do future architectures look like?

Figure 3 shows Moore's Law (the observation made in 1965 by Gordon Moore, co-founder of Intel): The number of transistors on a microprocessor would double approximately every 18 months. Meanwhile, memory densities (DRAM) and disk densities will continue to quadruple every three years. The gap between microprocessor and memory will be discussed in detail in section 1.2.

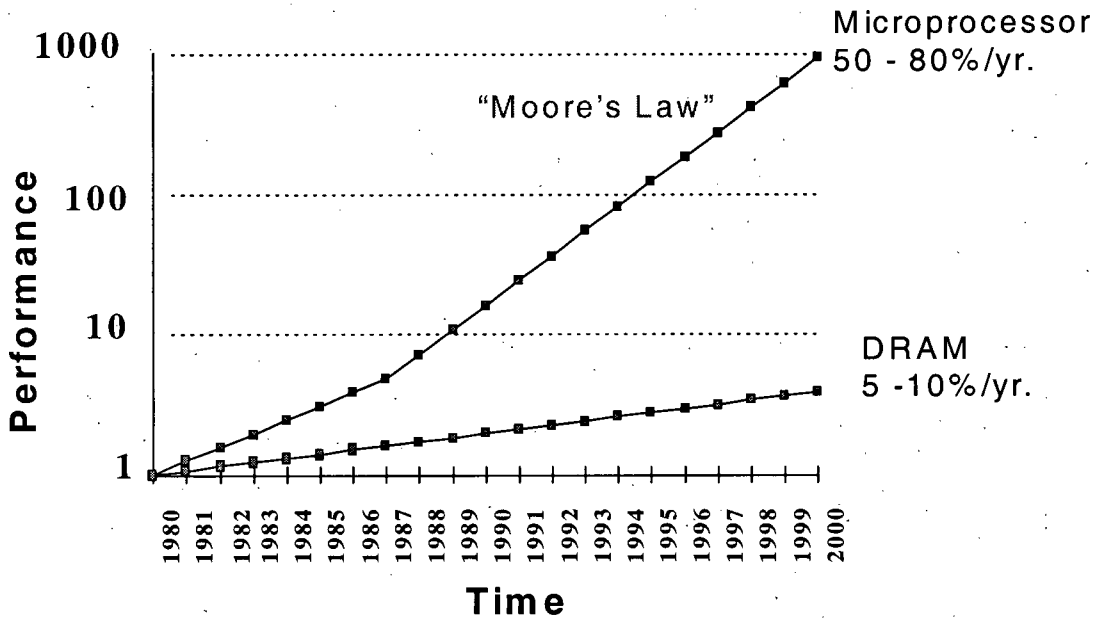


Figure 3. Comparison of the performance between microprocessor and DRAM according to Moore's Law [Patterson and Keeton '00] [Alexander & Kedem '95].

To date, Moore's Law has proven remarkably accurate even if the end of Moore's Law has been predicted so many times that rumors of its demise have become an industry joke. In reality, microprocessors have achieved a performance growth of 10,000 times

during the last two decades. Transistor count increased from 10,000 to 100,000 in the 1970s, and then increased up to 1 million in the 1980s; while clock frequency increased from 200KHz to 2MHz in the 1970s and up to 20MHz in the 1980s. In the 1990s, both transistor count and clock frequency achieved an increase of 20 to 30 times. Future billion-transistor chips in the 2000s will create machines that are much wider (issue more than four instructions at once) and deeper (have longer pipelines) [Lipasti & Shen '97].

According to Burger & Goodman ('97) and Patt et al. ('97), microprocessors will have more than one billion transistors on a single chip by 2010. As we discussed before, most of the current techniques for microprocessors have come from supercomputers. Moreover, some future techniques will be based on current ones such as instruction level parallelism. The future architectures surveyed by Burger & Goodman ('97) are:

- *Advanced Superscalar processors* that issue 16 to 32 instructions per cycle and *Superspeculative processors* that have wide-issue speculation;
- *Vector IRAM processors* couple vector processor execution with large, high-bandwidth, on-chip DRAM banks, which provide the vector units with sufficient bandwidth at a reasonable cost;
- *Chip multiprocessors* that place a number of processors (four to 16) on a single chip;
- *Raw processors* that implement parallel architectures with 128 tiles, very simple processors with reconfigurable functional logic.

Beyond the previous potential architectures, *Simultaneous multithreaded processors* and *Trace processors* are also included in the surveyed future architectures [Burger & Goodman '97].

1.2 Problem definitions and resolutions

The previous section provided an overview of high performance computing. This section will discuss two problem definitions intended to improve system performance for current and future microprocessors.

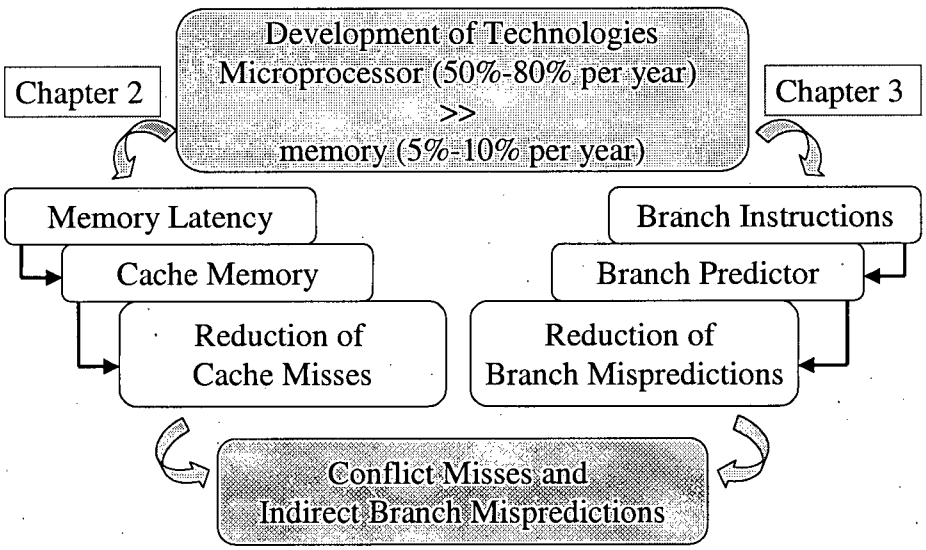


Figure 4. Problem definitions

In Figure 4, rapidly changing technologies are improving microprocessor execution speeds at a rate of 50% - 80% per year. In contrast, DRAM access time has developed at the much lower rate of 5% - 10% per year [Alexander & Kedem '95]. As the performance gap between microprocessor and memory increases dramatically, more and more programs will be limited in performance:

- by the memory latency and bandwidth of the system;
- by the branch instructions (control flow of the programs).

Latency is described as the total time memory requires to satisfy a request from the processor, and bandwidth as the rate of information transfer between the processor and memory that supports the required processing rate.

1) First Problem Definition (Left side of Figure 4): Cache Misses

Since the processor is much faster than the main memory, latency often causes the processor to go into one or more wait states. In order to solve the latency problem, a cache memory has been introduced as part of a memory hierarchy. The memory hierarchy combines a fast, small memory matched to the processor speed with slower and larger memories (level-two or main memory).

When a CPU does not find data it needs in a cache memory, a cache miss occurs. If a cache miss occurs, the CPU must wait until the needed data is retrieved from a lower level memory.

The impact on CPU performance caused by cache miss rates is:

CPU Execution time = IC * (CPI + (Memory stall clock cycle/Instruction))*

Clock cycle time

= IC * (CPI + (memory accesses per instruction) * MR * MP) *

Clock cycle time

Where, IC (Instruction Count), CPI (Cycles per Instruction), MR (Miss Rate), and MP (Miss Penalty).

*Relative CPU Execution time = CPU Execution time / (IC * Clock cycle time* CPI)*

If there are no memory stalls (perfect cache), then Relative CPU Execution time is 1.

Assume that CPI = 2, memory references per instruction = 1.33, Cache Miss Rate = 10%, and Miss Penalty = 50 cycles.

Then, Relative CPU Execution time = (2 + 1.33 * 0.1 * 50) / 2 = 4.33.

This Relative CPU Execution time shows that a CPU Execution time of 10% cache miss rate is 4.33 times longer than a CPU Execution time with a perfect cache (0% cache miss rate).

Figure 5 shows Relative CPU Execution time when the behavior of the cache (from 0% cache miss rate, perfect cache, to 10% cache miss rate) is included.

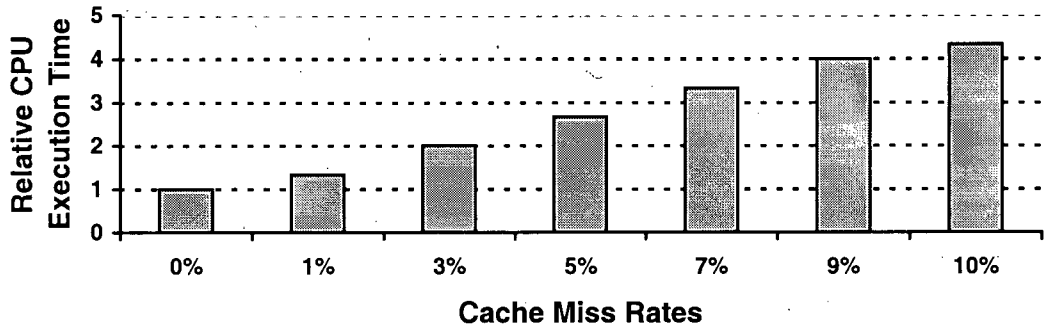


Figure 5. Relative CPU execution time by cache miss rates.

Without any memory hierarchy at all, the CPI would increase to 2.0 + 50*1.33 or 66.5 – a factor of over 33 times longer.

As the above example illustrates, cache behavior can have enormous impact on performance. Therefore, the efficiency of a cache memory depends on reducing cache misses and will be discussed in detail in chapter 2. This thesis defines reduction of cache misses as the first problem to be solved and in chapter 2 introduces a new cache scheme to reduce cache misses, focused on conflict misses due to the cache set overfilling, even though the cache as a whole may not be full.

2) Second Problem Definition (Right side of Figure 4): Branch Mispredictions

For current microprocessors, multi-instruction issues are a popular method of increasing system performance. Therefore, instruction cache misses can severely limit the performance of high-speed microprocessors. It has been observed that many instruction cache misses are caused by the control flow of programs.

Control flow is related to the branch instructions, which can be generally categorized into conditional or unconditional, and direct or indirect [Chang et al '97]. Since these branch instructions do not tend to fetch the next instruction in sequence, it is not possible to know the next instruction until a current instruction is executed. To overcome this obstacle, branch prediction schemes have been used for predicting and fetching the outcome of branches before they are executed. Therefore, if the prediction is wrong (branch misprediction), the processor needs to be stalled because as a result of flushing all the instructions incorrectly fetched, issued, and executed. This is referred to as branch penalty. Thus, without an appropriate branch predictor, the branch penalty can have a critical impact on overall system performance.

If branches are the only thing that cause stalls in a pipeline, the impact of CPU performance caused by branch penalty is:

CPU Execution time = IC * (CPI_{base} + branch frequency * branch penalty)*

Clock cycle time

Where, IC (Instruction Count), CPI_{base} (an ideal CPI without branch stalls in the pipeline), branch penalty (branch misprediction rate * misprediction penalty).

*Relative CPU Execution time = CPU Execution time / (IC * Clock Cycle time * CPI_{base})*

If there are no branch stalls (perfect branch predictor), Relative CPU Execution time is 1.

Assume that CPI_{base} = 1, branch frequency = 25%, branch misprediction rate = 20%, and misprediction penalty = 5 cycles.

Then, Relative CPU Execution time = (1 + 0.25 * 0.2 * 5) / 1 = 1.25.

This Relative CPU Execution time shows that a CPU Execution time of 20% branch misprediction rate is 1.25 times longer than a CPU Execution time with a perfect branch predictor (0% branch misprediction rate).

Figure 6 shows Relative CPU Execution time when the behavior of the branch predictor (from 0% branch misprediction rate, perfect branch predictor, to 40% branch misprediction rate) is included.

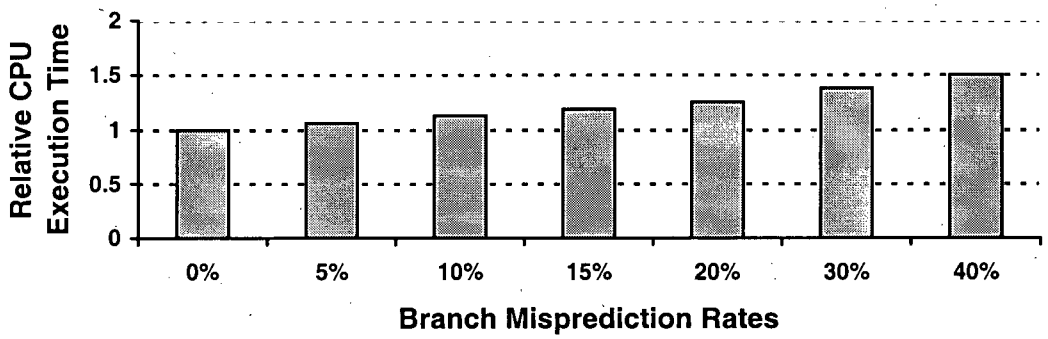


Figure 6. Relative CPU execution time by branch misprediction rates.

To reduce the branch penalty, there is a need to reduce branch mispredictions: direct mispredictions and indirect branch mispredictions. As object-oriented languages such as C++ and JAVA are widely used, more accurate branch predictors for multi-targets, which are called indirect branch predictors, are needed. This thesis also defines the reduction of branch mispredictions as the second problem to be solved, and introduces in chapter 3 a new branch predictor to reduce branch mispredictions focused on indirect branch mispredictions due to multi-targets.

1.3 General background

The previous section briefly describes the problems defined in this thesis. In this section, we discuss the general background of cache misses and branch mispredictions in more detail.

1.3.1 Cache Misses

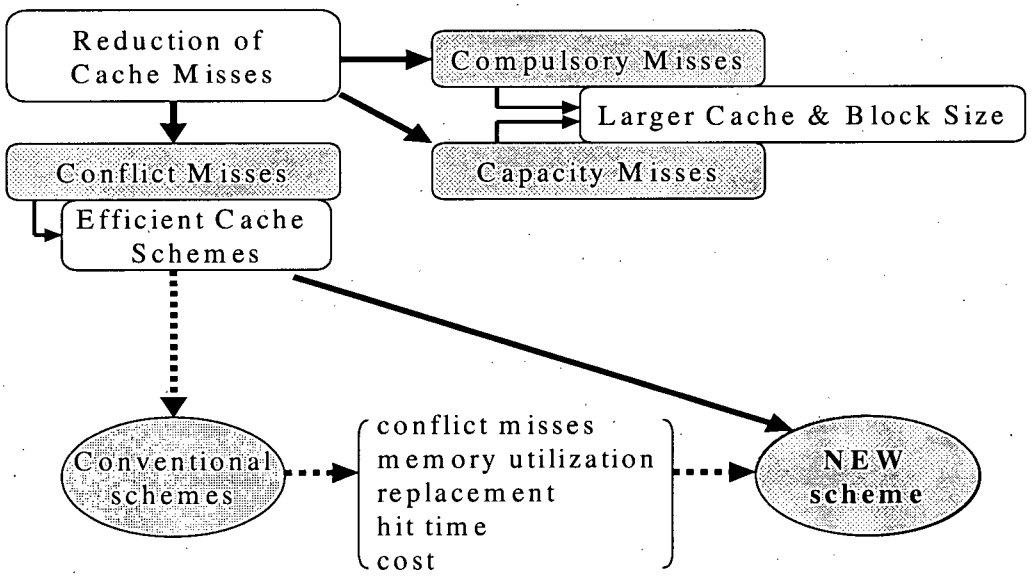


Figure 7. Reduction of Cache Misses (overview).

Figure 7 shows the problem of reduction of cache misses. There are three cache-miss types: compulsory, capacity, and conflict. Compulsory and capacity cache misses can be reduced by larger cache and block sizes. However, conflict misses are more complex than other cache misses and are critical to system performance. Figure 7 also suggests the problems which the conventional cache schemes have in regard to conflict misses, memory utilization, etc. In Chapter 2, we will discuss the conflict miss ratios of several

cache schemes (direct-mapped, 2-way skewed associative, etc.) compared to the fully-associative scheme which has no conflict misses, and also introduces a new cache scheme called the TAC (Thrashing-Avoidance Cache) scheme, which can reduce conflict misses effectively.

Cache Miss Types

Despite tremendous research efforts, current cache schemes make poor use of cache capacity. One of the drawbacks of conventional cache schemes is that they perform a myopic management of all memory references: if the reference misses, a new block is brought into the cache at the expense of replacing another [Sanches et al '97].

There are three cache-miss types - compulsory, capacity, and conflict [Patterson & Hennessy '96]:

- Compulsory misses: these are the first reference misses since a block must be brought into the cache the first time it is accessed;
- Capacity misses: if the number of active blocks is more than the cache can contain, capacity misses take place;
- Conflict misses: these misses take place because of limited or zero associativity, when blocks must be discarded in order to accommodate new blocks which are mapped to the same line in the cache. A conflict miss occurs when the replaced block needs to be accessed.

In case of compulsory misses, it is not possible to avoid these misses since the first access is to a block that is not in the cache. Fortunately, the frequency of these misses tends to be quite small compared to other cache misses.

Block Size	Cache Size				
	1K	4K	16K	64K	256K
16	15.05 %	8.57 %	3.94 %	2.04 %	1.09 %
32	13.34 %	7.24 %	2.87 %	1.35 %	0.70 %
64	13.76 %	7.00 %	2.64 %	1.06 %	0.51 %
128	16.64 %	7.78 %	2.77 %	1.02 %	0.49 %
256	22.01 %	9.51 %	3.29 %	1.15 %	0.49 %

Table 1. Actual miss rate versus block size for five different-sized caches. Note that for a 1-KB cache, 64-byte, 128-byte, and 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses [Patterson & Hennessy '96].

Table 1 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger cache and block sizes reduce compulsory misses since larger blocks take advantage of spatial locality. At the same time, the larger blocks increase any miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache size is small.

Basic Mapping Functions

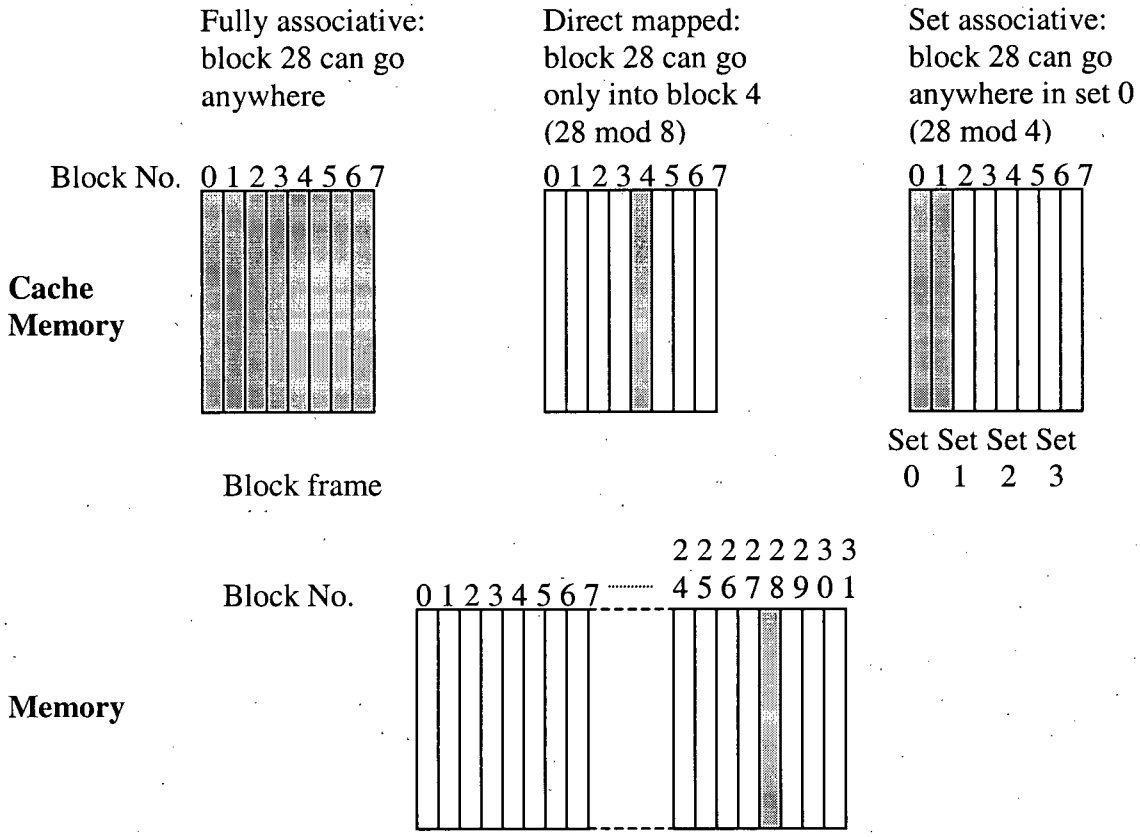


Figure 8. This cache example has eight block frames and memory has 32 blocks [Patterson & Hennessy '96].

The basic mapping functions can be categorized into the following types:

- If a block can be placed anywhere in the cache, the cache is said to be *fully-associative*;
- If each block has only one place it can appear in the cache, the cache is said to be *direct-mapped*. The mapping is usually (Block address) MOD (Number of blocks in cache);

- If a block can be placed in a restricted set of places in the cache, the cache is said to be *set associative*. A set is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by bit selection; that is, (Block address) MOD (Number of sets in cache). If there are n blocks in a set, the cache replacement is called n-way set associative.

Figure 8 shows that the restrictions on where a block is placed create three categories of cache organization. The set-associative organization shown has four sets with two blocks per set, and is called two-way set associative. Assume that there is nothing in the cache and that the block address needed identifies lower-level block 28. The three options for caches are shown left to right. In fully-associative, block 28 from the lower level can go into any of the eight block frames of the cache. With direct-mapped, block 28 can only be placed into block frame 4 (28 modulo 8). In two-way set associative, the block is to be placed anywhere in set 0 (28 modulo 4). With two blocks per set, this means block 28 can be placed either in block 0 or block 1 of the cache. The vast majority of processor caches today are direct-mapped, two-way set associative, or four-way set associative.

1.3.2 Branch Mispredictions

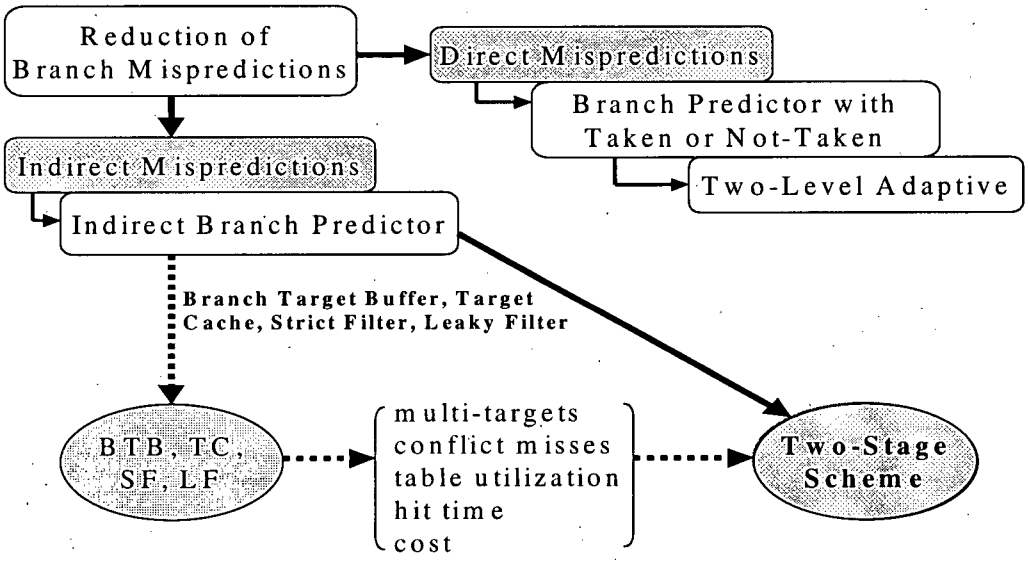


Figure 9. Reduction of Branch Mispredictions (overview).

Figure 9 shows methods for the reduction of branch mispredictions, which are categorized into direct and indirect branch mispredictions. Direct branches can be predicted with two-level branch predict schemes with hit ratios of up to 97%. However, it cannot be used for indirect branches which have more than one target. Chapter 3 explains indirect branch mispredictions in detail and discusses current indirect hybrid branch predictors such as Target Cache and Cascaded Predictor. These predictors work better than BTB-based predictors, which are used to predict for a single target such as direct branches, but they suffer from conflict misses in the first stage predictor and have inefficient update rules. Chapter 3 introduces a new indirect hybrid branch predictor called the GoStay2 predictor to improve the update rules and reduce conflict misses in the first stages.

Branch prediction is a key mechanism used to achieve high performance on multiple issue, deeply-pipelined processors. By predicting the branch outcome at the instruction fetch stage of the pipeline, ILP (Instruction Level Parallelism) can be exploited by providing a larger window of instructions [Kalamatianos & Kaeli '98].

Branch Classification

Branches can be categorized as conditional or unconditional and direct or indirect, resulting in four classes: conditional direct, conditional indirect, unconditional direct, and unconditional indirect. Of the four classes, prediction of conditional indirect branches are typically not implemented [Kalamatianos & Kaeli '98].

Conditional direct branches, which involve a condition, have two types: loop-closing conditional branches and other conditional branches. The loop-closing branches are backward branches that are taken for all but the last iteration of a loop. Other conditional branches are either taken or not taken, depending on whether the specified condition is true of false [Sima et al '97].

Unconditional direct branches, which are always taken, have three types: simple unconditional branches, branches to subroutines, and returns from subroutines. Simple unconditional branches do not save the return address, whereas branches to subroutines do. Returns from subroutines are dedicated unconditional branches performing a control transfer to the saved return address. In case of nested subroutines, while branching to and returning from the individual subroutines, the return addresses are saved and used in a last-in first-out (LIFO) manner.

A *conditional/unconditional direct* branch has a statically specified target that points to a single location in the program, whereas an *unconditional indirect* branch has a dynamically specified (i.e. computed) target that may point to any number of locations, multi-targets, in the program. Indirect branches with multi-targets are harder to predict accurately than single-target direct branches.

Driesen & Holzle ('98) also classified branches according to the number of different targets encountered in a program run (SPECint95 and object oriented languages): one target, two targets, and more than two targets. Branches with only one target constitute 67% of all branches; 18% of all branches jump to two targets and branches with three or more targets constitute 15% of all branches.

Branch Predictors

There are several types of branch predictors such as one-level, two-level, hybrid, etc. For the one-level predictor, a BTB (branch target buffer) is commonly used. BTB is a cache that contains the address of the branch instructions and their target addresses. The BTB is accessed in the fetch stage to predict the state of a branch instruction. If a hit occurs, then the current instruction is a taken branch. The PC (program counter) is loaded with the target address from BTB, and fetching starts from the new PC. For indirect branch, the taken address is the last computed target for the indirect jump. Unfortunately, BTB-based prediction schemes perform poorly for indirect jumps [Chang et al '97].

The two-level branch predictor uses two levels of history to make branch predictions [Yeh and Patt '92]. The first-level of history records the outcomes of the most recently

executed branches and the second-level history keeps track of the more likely direction of a branch when a particular pattern is encountered in the first level history. The 2-level branch predictor uses one or more k-bit shift registers, called branch history registers, to record branch outcomes of the most recent k branches. It uses one or more arrays of 2-bit saturating up-down counters, called a Pattern History Table (PHT), to keep track of the more-likely direction for branches. The lower bits of the branch address select the appropriate PHT and the value in the Branch History Register (BHR) selects the appropriate 2-bit counter to use within that PHT. There are many variations of two-level predictor. In Chapter 3, we will discuss various branch predictors in detail.

According to Sima et al ('97), the prediction accuracy of BTB is less than 70% in the processor MC88110. In order to improve the prediction accuracy of simple BTB, more complex hardware such as a two-level adaptive BTB, which can detect more varied branch execution sequences and treat them individually, has been proposed to take advantage of the relationship between nearby branches to improve its branch prediction accuracy. Even if the misprediction rate is less than 10%, the residual misprediction penalty that these programs incur still deteriorates processor performance significantly.

Branch Misprediction Penalty

In Figure 10, Bondi et al ('96) show the total CPI (Cycles Per Instruction) for the model classified as normal processing, branch misprediction penalty, and memory access wait cycles (imperfect cache). They evaluated x86 traces with the performance model of a microprocessor design comprising a moderate-depth pipeline, 2-bit branch predictor, 4

integer execution resources, and on-chip instruction and data caches. The SPECint92 traces were generated by running the subject program on a PC under DOS after compilation with the gcc compiler from DJGPP (one of SPECint92 benchmark programs).

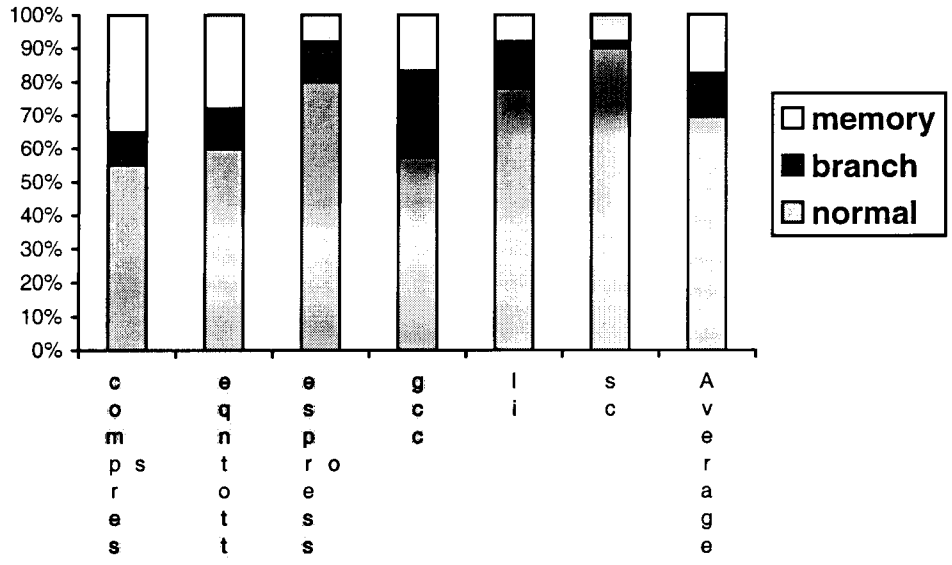


Figure 10. Components of Total CPI (Cycles Per Instruction) [Bondi et al. '96]

Of the total processing expended, normal processing consumes about 70%, branch consumes about 13% and memory access consumes about 17%. Together, branch penalties and memory waits waste about 30% of the overall processing effort. So if the branch misprediction penalty and memory access wait cycles can be reduced further, the system performance can be improved substantially. For example, if the branch accuracy

rate is increased from 95.2% to 96.0%, then the misprediction rate can be reduced up to 17.6%. If the recovery time from misprediction is reduced, it also improves the overall CPI.

According to Bondi et al ('96), mispredicted branch instructions are categorized into two types: branches that are repeatedly mispredicted over program life, and branches that are mispredicted just once over program life. They showed that branches that have been previously mispredicted cause most mispredictions. This behavior suggests that there is a need to hold the flushed branch instructions caused by conflict misses in a specific cache memory in the processor.

1.4 Contributions and summary

As object-oriented languages are widely used, procedure calls are increasing frequently in application programs, causing a significantly increased number of conflict misses in the instruction flow. Basically, the instruction flow has several problems to solve: conflict misses in the instruction cache memory, conditional or unconditional branch throughput, direct or indirect branch prediction, and misprediction penalty. Current high performance architectures such as superscalar processors use branch prediction to speculatively execute instructions beyond an unresolved branch. If the branch is mispredicted, this work is lost, and execution must restart right after the branch instruction.

As we discussed in section 1.2, there is a need to reduce cache misses and branch mispredictions for improving system performance. The contributions of this thesis lie in the fact that:

1. By developing a new cache scheme called the TAC (Thrashing-Avoidance Cache)
   - Cache miss rates can be reduced significantly compare to other conventional cache memory schemes.
   - Since the TAC has almost the same hardware complexity as n-way set-associative, it is possible to increase system performance with the same hardware cost as n-way set-associative.
   - As small on-chip cache memory is popular, there is a need to have more efficient memory storage management than n-way set-associative. The TAC provides this by using sophisticated mapping functions.
   - The TAC scheme can be applied to the techniques for HPC in regard to instruction flows.
2. By developing a new indirect branch predictor called the GoStay2
   - Indirect branch mispredictions can be reduced significantly compared to other conventional indirect branch predictors.
   - Since the GoStay has almost the same hardware complexity as the other branch predictors, it is possible to increase system performance with the same hardware cost as the others.
   - The GoStay2 can increase instruction level parallelism by improving update rules for the indirect branch predictions.

There are four chapters and one appendix in this thesis. They include:

Chapter 1, Overview and summary, describes high performance computing, which is divided into three parts including goals, architectures, and techniques. Moore's Law is

discussed in order to understand the future trend of processors. From Moore's Law, we outlined problem definitions such as cache misses and branch mispredictions, which degrade system performance because of the gap between processors and memory. General background for cache misses and branch mispredictions are discussed, and a new cache scheme and branch predictor are suggested;

Chapter 2, Cache misses, gives an overview and problems of conventional cache schemes and introduces a new cache scheme called the TAC (Thrashing-Avoidance Cache). Through the experimental results, it is shown that the TAC schemes reduce conflict misses better than conventional cache schemes;

Chapter 3, Branch Mispredictions, explains branch mispredictions caused by direct and indirect conditional branches and discusses current branch predictors that were recently proposed to reduce indirect branch predictors. Since those branch predictors have an inefficient update rule, a new branch predictor called the GoStay2 predictor is introduced for improving branch prediction rates. Through experiments, it is shown that the GoStay2 works better than other indirect branch predictors such as Cascaded predictors or Target Cache;

Chapter 4, Conclusion and Future Research, summarizes the experimental results for the TAC scheme and GoStay2 predictor compared to conventional schemes and suggests future research in regard to reducing memory latencies and speculative work;

Appendix A, Experimental results for TAC schemes, gives detailed tables of the experimental results for TAC schemes.

Chapter 2 Reduction of instruction cache misses

Due to the increased complexity of application programs over the past decade, object-oriented languages are replacing traditional languages as a result of convenient code reusability and maintainability. However, it has also been observed that the run-time performance of object-oriented programs can be improved by reducing the impact caused by instruction cache misses. This thesis presents a new cache scheme called TAC (Thrashing-Avoidance Cache), which can effectively reduce instruction cache misses caused by procedure call/returns. The TAC scheme employs N-way banks and XOR mapping functions. The main function of the TAC is to place a group of instructions separated by a call instruction into a bank according to the Bank Selection Logic (BSL) and Bank-originated Pseudo-LRU replacement policy (BoPLRU). After the BSL selects a bank initially on an instruction cache miss, the BoPLRU will determine the final bank for updating a cache line as a correction mechanism. These two mechanisms can guarantee that recent groups of instructions exist in each bank safely. A simulation program, TACSim, has been developed by using Shade and Spixtools, provided by SUN Microsystems, on an ultra SPARC/10 processor. Our experimental results show that TAC schemes reduce conflict misses more effectively than skewed-associative caches for both C (9.29% improvement) and C++ (44.44% improvement) programs on L1 caches. In addition, TAC schemes also allow for a significant miss reduction on Branch Target Buffers (BTB).

2.1 Introduction

For current microprocessors, multi-instruction issues are a popular method of increasing system performance. Therefore, instruction cache misses can severely limit the performance of high-speed microprocessors.

Several researchers have shown that many instruction cache misses are caused by the frequent procedure call/returns in object-oriented languages.

		C++ mean (12 C++ programs)	C mean (SPECint92)	Ratio (C++/C)	Description
Type		Object Oriented	Traditional		
Call/Return Frequency		4.6%	0.7%	6.7	Procedure calls and returns
Basic block		4.8	5.9	0.8	Instructions per block size
Function size	Dynamic	48.7	152.8	0.3	During program run-time
	Static	27.3	44.3	0.6	Property of program itself
Inst. Cache miss rate (Direct mapped, 32byte line)	4 K	5.83	3.49	1.67	C++ programs tend to perform many calls to small functions and benefit less from the spatial locality of larger cache blocks. Average Ratio (C++/C):1.95
	8 K	3.98	2.32	1.72	
	16 K	2.47	1.18	2.09	
	32 K	1.37	0.59	2.32	
Data cache miss rate (Direct mapped, 32byte line)	4 K	13.98	13.09	1.06	Since the miss rates are quite similar, there is little room to improve data cache features. Average Ratio (C++/C):1.02
	8 K	9.20	9.08	1.01	
	16 K	6.35	6.43	0.98	
	32 K	4.42	4.31	1.03	

Table 2. Behavioral differences between C and C++ Programs [Calder et al '94]

In Table 2, Calder et al ('94) showed that object-oriented programs (C++) execute almost seven times more calls (4.6 % versus 0.7 %) and have smaller function sizes (48.7 versus 152.8 instructions/function) than traditional programs (C). While C programs execute large monolithic functions to perform a task, C++ programs tend to perform

many calls to small functions. Thus, C++ programs benefit less from the spatial locality, and suffer more from function call overhead.

The smaller function size of C++ programs is another cause of poor instruction cache misses. According to Calder et al ('94), programs executing a small number of instructions in each function, such as C++, may suffer from instruction cache conflicts. For example, two mutually recursive functions may be aligned to the same cache memory addresses and constantly displace each other from the cache. C programs execute more instructions per function invocation, meaning that more work is done within a particular function.

Holzle & Ungar ('94) also showed that for instruction cache behavior the miss ratios of object-oriented programs are significantly higher for most cache sizes and that the median miss ratio is 2 – 3 times higher than traditional programs. Meanwhile, Calder et al ('94) and Holzle & Ungar ('94) observed that the data cache misses for both programs were seen to be similar. So this thesis has focused on developing an effective cache scheme to reduce the instruction cache misses of object-oriented programs, which can be much higher than traditional programs because of the frequent call/returns.

In general, if a cache size is less than 32KB, conflict misses can degrade system performance significantly. For example, for a direct-mapped cache, conflict misses are about 60% of the total cache misses of a small-sized cache of 8KB [Gonzalez et al '97]. If we do not want to increase the cache size, we need to design a small-sized, low-cost cache scheme to improve the cache miss ratio by reducing only the conflict misses which are mainly caused by call/returns. This thesis presents a new cache scheme called TAC

(Thrashing-Avoidance Cache), which can effectively reduce instruction cache misses caused by call/returns.

This chapter is organized as follows: Section 2.2 explains cache misses and skewed-associative caches; section 2.3 presents a new instruction cache scheme called TAC (Thrashing-Avoidance Cache); section 2.4 describes simulation methodology and benchmark programs; section 2.5 presents our simulation results; and section 2.6 provides our chapter conclusions.

2.2 Cache Misses

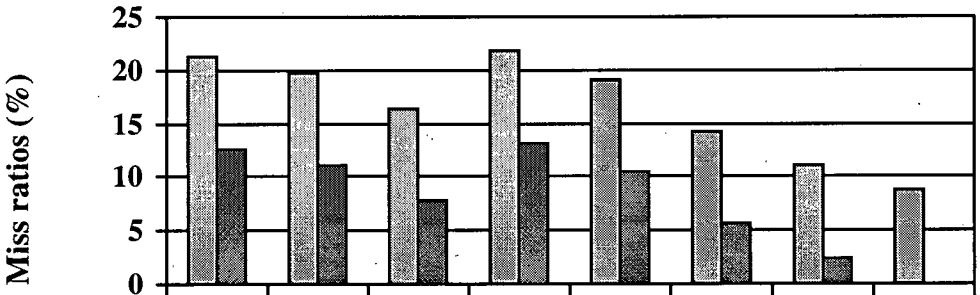
As we discussed in chapter 1, there are three types of cache misses namely: compulsory, capacity, and conflict misses. In this section, several conventional cache schemes are compared for determining the most effective conventional cache scheme for reducing conflict misses.

2.2.1 Total miss ratios vs. conflict miss ratios

Gonzalez et al ('97) generated the miss ratios for several cache schemes as shown in Figure 11: direct-mapped, 2-way set-associative, 4-way set-associative, hash-rehash, column-associative, victim, and 2-way skewed-associative. They obtained the results in Figure 11 by using the SPEC95 benchmark suite and by implementing a cache memory (8 kilobytes capacity and 32 bytes per line).

For comparison, the miss ratio of a fully-associative cache is shown in the last column. For each organization, the difference between its miss ratio and that of a fully-associative cache represents the conflict miss ratio. For example, the 'direct-mapped' cache has a

miss ratio of '21.32' in Figure 11. Here, '21.32' means the total miss ratio (compulsory + capacity + conflict) while '12.61' is the conflict miss ratio which is computed as (total miss ratio for a scheme – total miss ratio for the fully-associative scheme).



	direct	2-way	4-way	hash-rehash	col-assoc.	victim	2-way skew	fully-assoc.
total miss ratios (%)	21.32	19.76	16.42	21.87	19.11	14.27	11.05	8.71
conflict miss ratios (%)	12.61	11.05	7.71	13.16	10.4	5.56	2.34	0

Various cache schemes

Figure 11. Miss ratios (%) of the various cache schemes [Gonzales et al '97].

From the results in Figure 11, the hash-rehash scheme has a miss ratio similar to that of a direct-mapped cache. Although both have similar access times, the hash-rehash scheme requires two cache probes for some hits. Hence, the direct-mapped cache will be more effective. The victim cache scheme removes many conflict misses and it outperforms a 4-way set-associative cache. The 2-way skewed-associative cache offers the lowest miss ratio of the existing schemes and is significantly lower than a 4-way set-associative cache [Gonzalez et al '97].

Figure 12 shows how conflict misses can happen in a cache memory. It is assumed that there are 10 instructions (A, B... X, and Y) as an assembly code program in Figure 12, which include two procedure calls (B to H and I to X) and two returns (Y to J and J to C). It is also assumed that (B, H), (C, I, X), and (D, J, Y) have the same set address (cache

memory index), namely 004, 008, and 00C, and that there is a stack register for the two return addresses. The arrows in Figure 12 show instruction flows for executing this program.

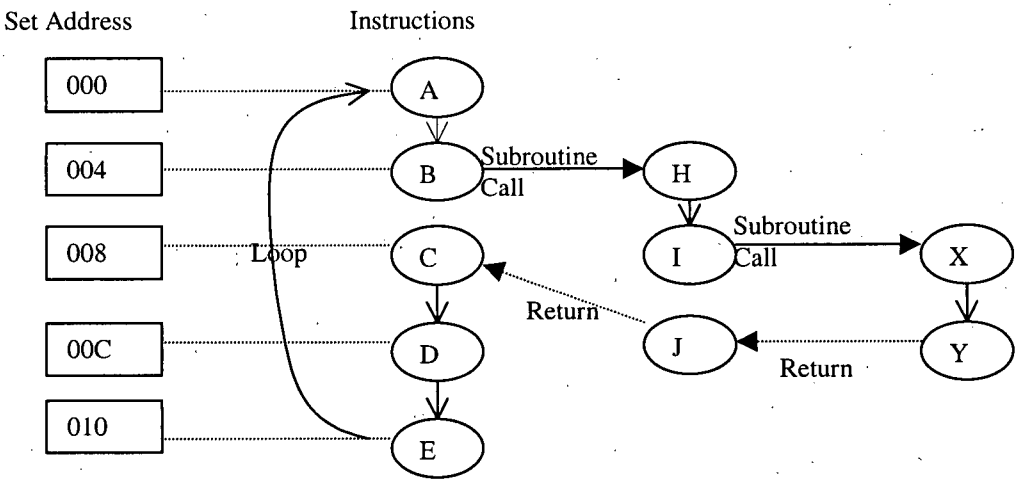


Figure 12. An example of instructions with two procedure calls.

Figure 13 shows the contents of a direct-mapped cache during execution of the two loops of code shown in Figure 12. In the case of the set address '008', there are two conflict misses with three memory accesses (to main memory) in the first loop and three conflict misses with three memory accesses in the second loop.

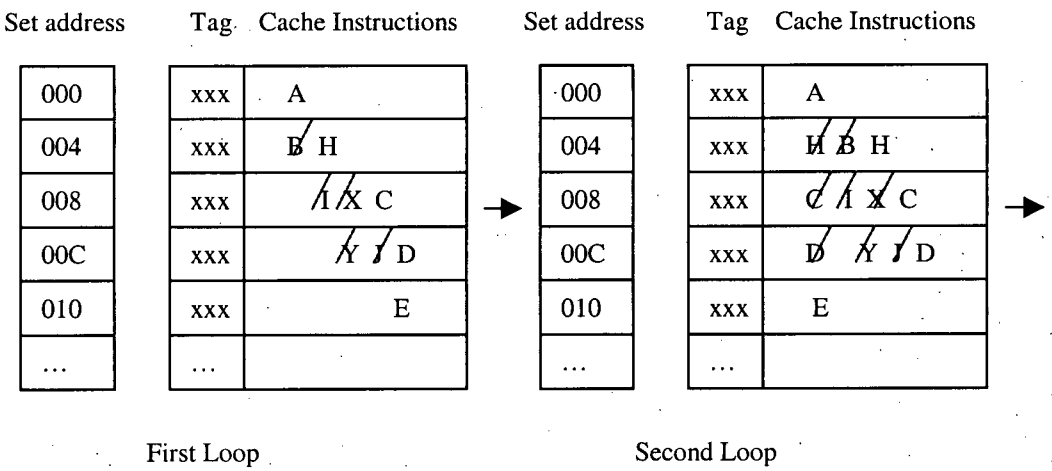
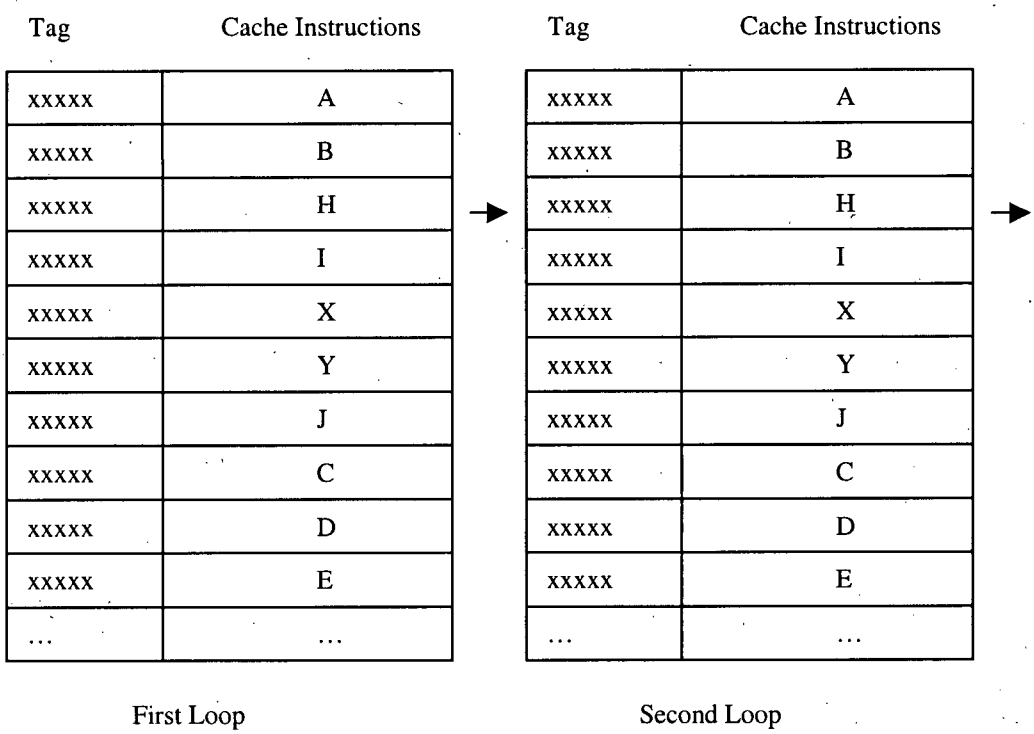


Figure 13. Execution of the code shown in Figure 12 in a direct-mapped cache.

For the direct-mapped cache in Figure 13, problems occur when alternating memory references point to the same set-address. Each reference causes a cache miss (conflict) and replaces the entry just replaced, causing a lot of overhead. The popular word for this is thrashing. When there is a great deal of thrashing, a cache can be more of a liability than an asset because each cache miss requires that a cache line be refilled - an operation that moves more data than merely satisfying the reference directly from main memory [Handy '93]. However, the direct-mapped cache has advantages of simplicity of memory access and hit time.

Figure 14 shows the contents of a fully-associative cache which has no conflict misses.



(b) Fully-associative cache.

Figure 14. Execution of the code shown in Figure 12 in a fully-associative cache.

For the fully-associative cache in Figure 14, any block (i.e., instruction) from the main memory can be placed anywhere in the cache. After being placed in the cache, a given block is identified uniquely by its main memory block number and referred to as the tag, which is stored inside a separate tag memory in the cache. The fully-associative cache makes the most flexible and complete use of its capacity, storing the blocks where it needs to, but there is a penalty to be paid for this flexibility: the tag memory must be searched in its entirety for each memory reference. Moreover, it is more expensive in terms of gates than other access-by-address memories, because of the need to do simultaneous bit-by-bit comparisons of all bits in the memory [Heuring & Jordan '97].

To reduce memory stalls effectively, there is a need to have a sophisticated form of cache memory, which has:

- less conflict misses;
- simplicity (access-by-address);
- faster hit time;
- efficient cache memory storage management; and
- low hardware costs.

2.2.2 Skewed-associative caches

In the previous section, Gonzalez et al ('97) showed that a 2-way skewed-associative cache offers the lowest miss ratio, and is significantly lower than that of a 4-way set-associative cache. Therefore, this scheme is discussed in detail in this section.

Skewed associative caches have been previously proposed by Seznec ('93). An N-way skewed-associative cache consists of N distinct banks that are accessed simultaneously with different mapping functions. For example, Figure 15 shows that a 2-way skewed-associative cache consists of two banks of the same size that are simultaneously accessed with two different mapping functions. That means a memory block at address 'd' may be mapped onto physical line f_0 (d) in bank 0 or onto f_1 (d) in bank 1, where f_0 and f_1 are different mapping functions.

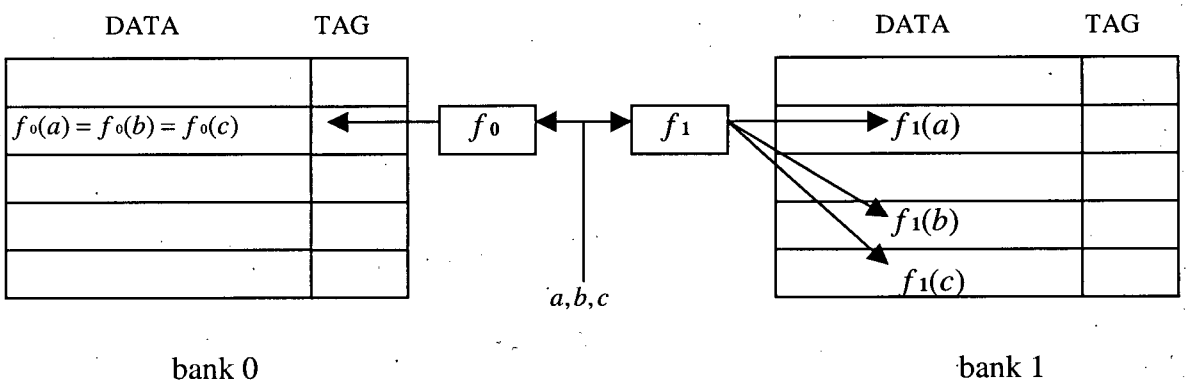


Figure 15. a, b, and c compete for the same location in bank 0, but can be present at the same time, as they do not map to the same location in bank 1 [Seznec '97].

Mapping functions

Bodin & Seznec ('95) presented skewing functions that are obtained by XORing a few bits in the address of a memory block. Let a skewed associative cache be built with 2 or 4 cache banks, each one consisting of 2^n cache lines of 2^c bytes, and let σ be the perfect-shuffle on n bits, so that the data block at memory address $A_3 2^{c+2n} + A_2 2^{n+c} + A_1 2^c$ may be mapped:

1. on a cache line $A_1 \oplus A_2$ in cache bank 0

2. or on a cache line $\sigma(A_1) \oplus A_2$ in cache bank 1

3. or on cache line $\sigma^2(A_1) \oplus A_2$ in cache bank 2 (on a 4-way)

4. or on cache line $\sigma^3(A_1) \oplus A_2$ in cache bank 3 (on a 4-way)

Replacement policies

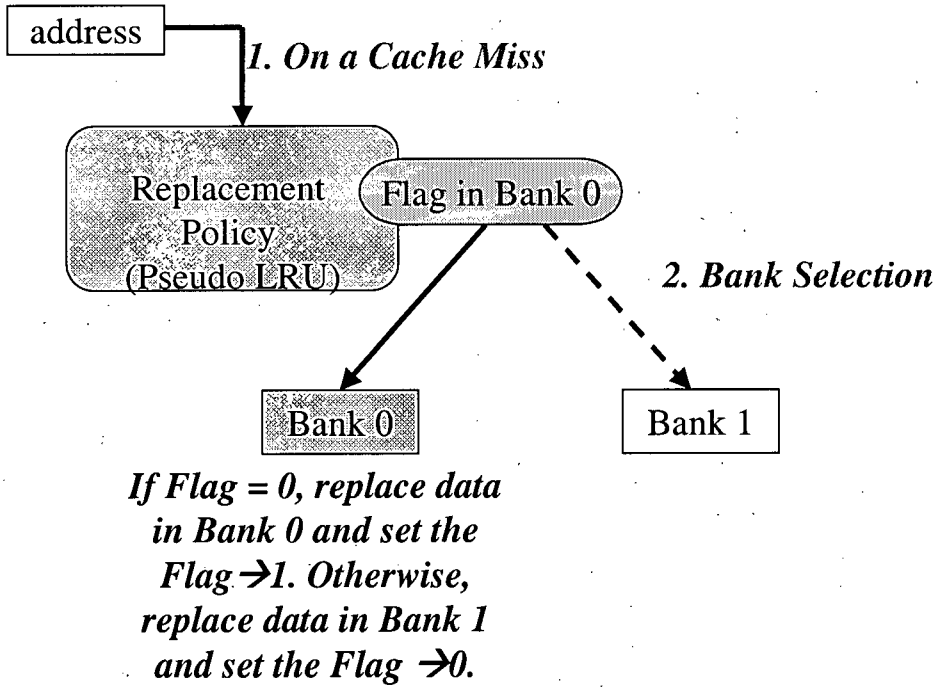


Figure 16. One of replacement polices, PLRU, for a 2-way skewed-associative cache.

Figure 16 shows that a 2-way skewed-associative cache uses a Pseudo-LRU (Least Recently Used) replacement policy by associating a one-bit flag to each line in bank 0 when a miss occurs on a cache [Seznec '97]:

- A flag bit is associated with each line in bank 0: when the line is indexed, the flag bit is set when the data is in bank 0 and reset when the data is in bank 1;
- On a miss, the flag of the line selected in bank 0 is read: when this flag is 1, the missing line is written in bank 1, otherwise the missing line is written in bank 0.

2.3 Thrashing-Avoidance Cache (TAC)

In the previous section, several cache memory schemes were investigated in detail: a direct-mapped scheme was shown to have the advantages of fast cache hit time and simplicity, but it has the problem of conflict misses that can adversely affect system performance. Using a fully-associative scheme can solve the conflict misses, but it is too expensive for implementation and inefficient for accessing to memory references. Even though a 2-way skewed-associative scheme partially resolves these problems, it still has an inefficient replacement policy for frequent procedure call/returns, which can increase conflicts for certain locations in a cache memory.

There are two main reasons for designing a new instruction cache memory:

- As technology changes, smaller on-chip L1 caches (less than 32 Kbytes) have replaced large external caches (greater than 256 Kbytes);
- As object-oriented languages become more widely used, procedure calls tend to increase in application programs, causing an increasing number of conflict misses.

Thus, there is a need to have a new cache memory scheme to reduce instruction cache misses focused on reducing thrashing conflict misses (i.e., a commonly used location is displaced by another commonly used location in a cycle).

2.3.1 An overview of a TAC scheme

If the cache size is relatively small, conflict misses can degrade system performance significantly [Gonzales et al '97]. Figure 17(a) shows that, in a conventional cache scheme, individual instructions (*A or B*) are placed or replaced into cache memory according to a mapping function and replacement policy on a cache miss. A conventional cache scheme works well for reducing conflict misses for traditional programs but not for object-oriented programs since traditional programs have fewer calls and larger function sizes than object-oriented programs (refer to section 2.1).

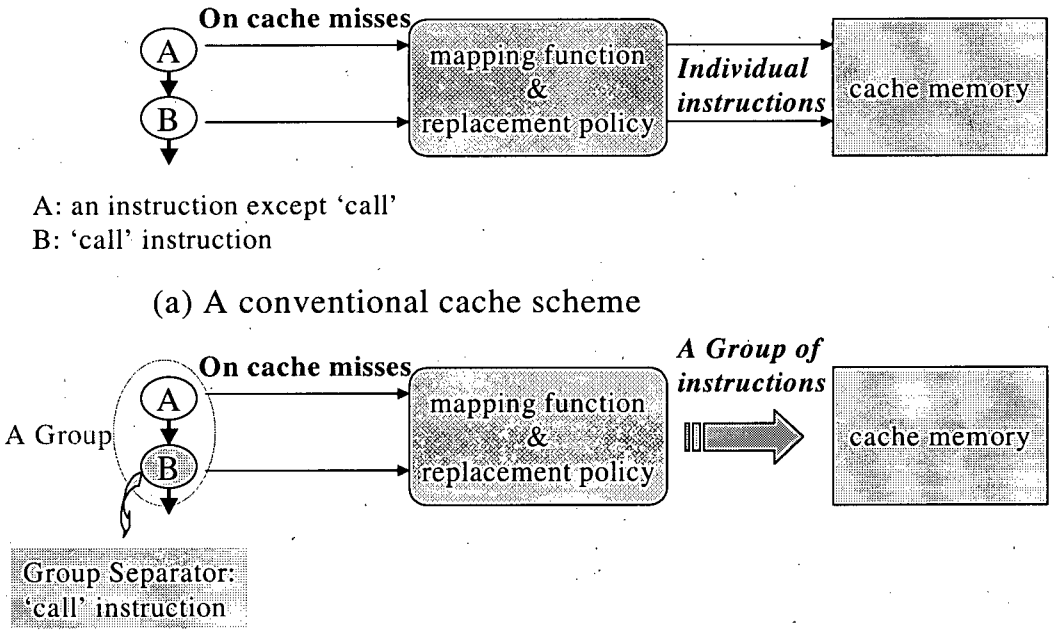


Figure 17. The basic operations of a conventional cache scheme and a TAC scheme.

Figure 17(b) shows the basic operations of a TAC scheme, which can reduce conflict misses effectively for object-oriented programs by grouping instructions. In Figure 17(b), a group of instructions (*A and B*) separated by call instruction (*B*) are placed and replaced into cache memory according to a mapping function or replacement policy on a cache

miss. Our measurements shows that grouping instructions benefits more from localities than individual instructions in both traditional and object-oriented programs.

A TAC scheme is built with N distinct banks. Since Gonzales et al ('97) showed that XOR mapping functions work well for reducing conflict misses, TAC employs XOR mapping functions (refer to section 2.2.2) for accessing the instruction cache memory.

On a cache miss, data must be fetched from a lower level memory according to the XOR mapping functions and replacement policies, the Bank Selection Logic (BSL, refer to section 2.3.2) and Bank-originated Pseudo LRU replacement policy (BoPLRU, refer to section 2.3.3). The BSL selects a bank initially according to the number of call instructions and the BoPLRU determines a final bank according to the replacement policy by using a flag.

The BSL and BoPLRU can guarantee that recent groups of instructions exist in each bank safely. So if the frequency of call/returns is increased, the TAC scheme works well since the manageable size of an instruction group is smaller. For example, if the average number of instructions per call of an object-oriented program is 40 and that of a traditional program is 100, then the TAC scheme of an object-oriented program will work better than a traditional one for limited cache sizes.

In a TAC scheme, each cache line consists of tag, data, and flag. The tag word consists of an address tag and some other status tags. The bit length of the flag is determined by the N distinct banks; that is, an n-bit flag represents 2^n banks or an N-Way ($N = 2^n$) cache scheme. For convenience, this thesis represents the cache line of a TAC scheme as just a flag and data throughout this paper and omits the tag part.

2.3.2 Bank Selection Logic (BSL) – Initial Bank Selection

The function of the Bank Selection Logic (BSL) is to select a bank initially on a cache miss according to a fixed frequency of the procedure call instructions. The BSL employs an x-bit counter for counting the frequency of call instructions. The x-bit counter will be increased by one whenever a fetched instruction proves to be a call instruction. An n-MSBs (n-Most Significant Bits) of the x-bit counter represents a selected bank for each instruction. Each bank can be selected for every 2^{x-n} procedural calls. For example, if x = 2 and n = 1, then there are two banks ($2^n = 2$) and a bank is switched to the other bank for every two procedure calls ($2^{x-n} = 2$). A group of instructions terminated by a procedure call can be placed into the same bank through the BSL (Bank Selection Logic) and XOR mapping functions. The goal of the BSL is to help each bank to share instructions equally according to the occurrence of procedure call instructions.

As an example, Figure 18 shows how a 2-bit counter (x = 2 and n = 1) in the BSL works with the flow of example instructions in Figure 18. The left side of Figure 18 shows the flow of instructions. Each call instruction works as a separator for grouping instructions. For a group of instructions, the next call instruction becomes the last one in the group. In Figure 18, it is assumed that there are cache conflicts in (B, H) and (I, X). The detailed operations of the 2-bit counter in the BSL on the right side of Figure 18 are:

- Instruction A is fetched. On a cache miss, the flag of the selected line in bank 0 is read. A is not a call instruction, so there is no change in the 2-bit counter (+ 0);
- Instruction B is fetched. On a cache miss, the flag of the selected line in bank 0 is read. B is a call instruction, so one is added to the 2-bit counter (+ 1);

- Instruction H is fetched. On a cache miss, the flag of the selected line in bank 0 is read. H is not a call instruction, so there's no change in the 2-bit counter (+ 0).
- Instruction I is fetched. On a cache miss, the flag of the selected line in bank 0 is read. I is a call instruction, so one is added to the 2-bit counter (+ 1);
- Instruction X is fetched. On a cache miss, the flag of the selected line in bank 1 is read. X is not a call instruction, so there's no change in the 2-bit counter (+ 0);
- Instruction Y is fetched. On a cache miss, the flag of the selected line in bank 1 is read. Y is not a call instruction, so there's no change in the 2-bit counter (+ 0); and
- Instruction J is fetched. On a cache miss, the flag of the selected line in bank 1 is read. J is not a call instruction, so there's no change in the 2-bit counter (+ 0).

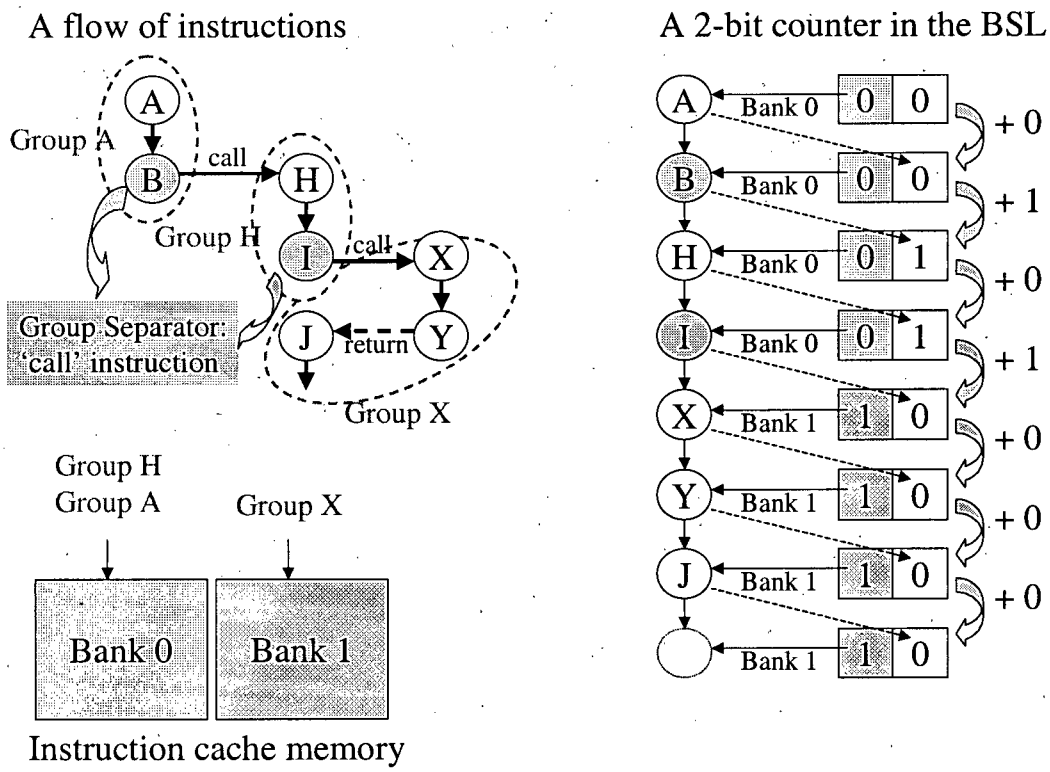


Figure 18. The operation of the BSL (2-bit counter, 2-way) according to a flow of instructions. Conflicts in (B, H) and (I, X).

Grouping instructions

Figure 18 shows that each 'call' instruction works as the selector of the distinct bank for instructions following the 'call' instruction. That means those instructions after the call instruction can be grouped together since they access the same bank on a cache miss until another call adds one to the n-bit counter in the BSL.

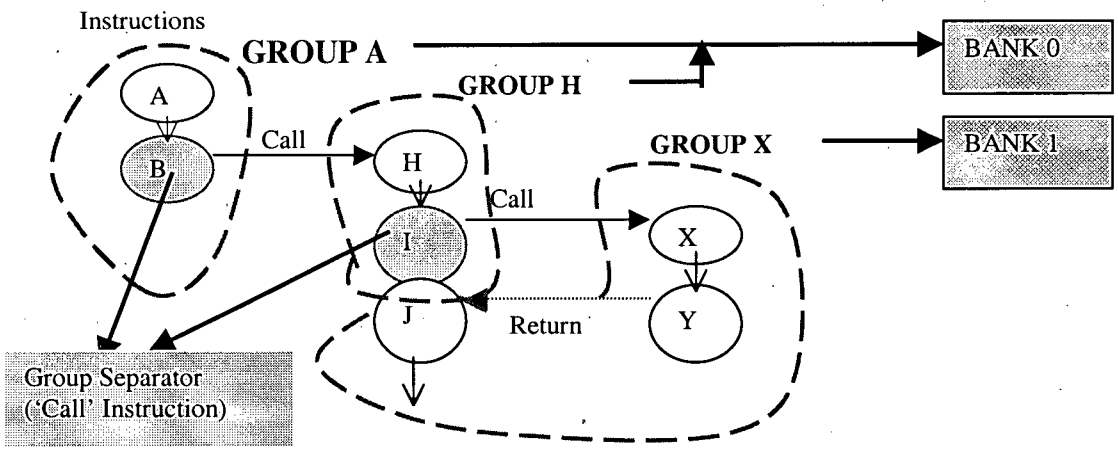


Figure 19. An example for the grouping instructions in a 2-way TAC scheme.

In Figure 19, the 'call' instruction (B) works as a separator for grouping instructions. The H instruction followed by the B instruction leads the group of instructions. The next 'call' instruction (I) is the last one in that group of instructions. Therefore, the group of instructions separated by the B instruction are {H, I}. In the same way, it is possible to group instructions in Figure 19 into {A, B}, {H,I}, and {X, Y, J, ..}. If each group is named after the leading instruction, there are three instruction groups such as group A, group H, and group X. Figure 19 shows that group A and group H access bank 0 and group X accesses bank 1 on a cache miss.

Consequently, there are three important properties in regard to grouping instructions in the following ways:

- Each 'call' instruction works as a separator for grouping instructions;
- The instruction following any 'call' instruction leads the group of instructions; and
- The next 'call' instruction terminates that group of instructions and works as a separator for the next group of instructions.

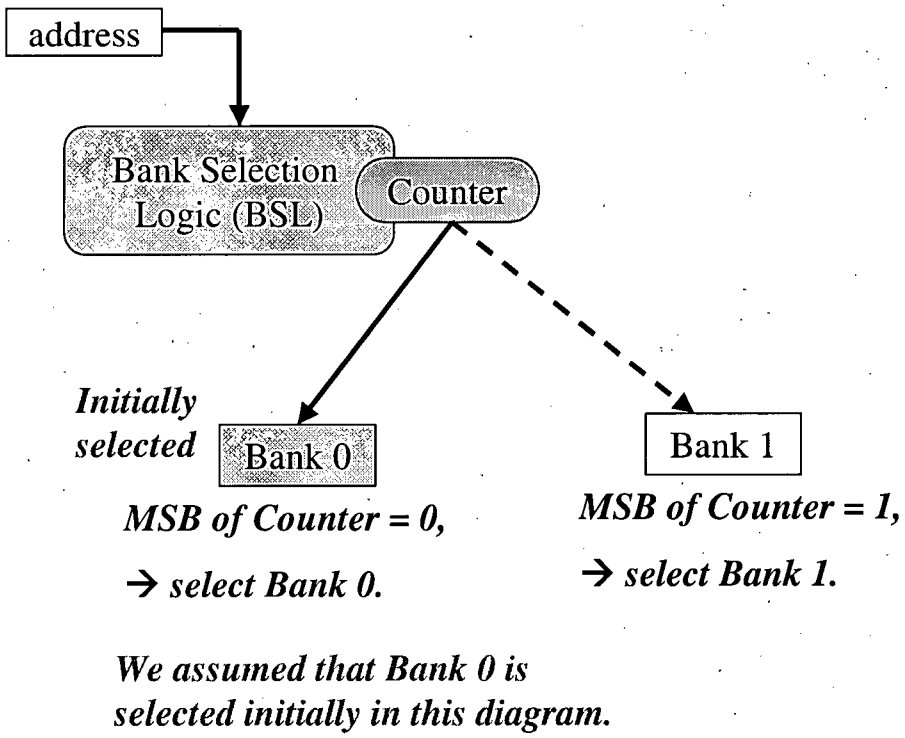


Figure 20. Initial bank selection of BSL for a 2-way TAC scheme.

As an example, the BSL operation of the 2-bit counter in a 2-way TAC scheme is shown in Figure 20: On a cache miss, the BSL initially selects a bank according to the value of the counter. If the MSB (Most Significant Bit) of the counter is 0, then bank 0 is selected. Otherwise, bank 1 is selected.

2.3.3 Bank-originated Pseudo LRU Replacement Policy (BoPLRU) – Final Bank Selection

After the BSL selects a bank on a cache miss, the BoPLRU will determine the final bank for updating a line as a correction mechanism by checking the flag for the selected cache line.

==========

The BSL selects a bank initially (say, initial bank).

If a 2-way TAC scheme, which has two banks

If 'the flag = 0' of the initial bank

Replace data of the other bank.

Set the flag of the initial bank to 1.

If' the flag = 1' of the initial bank

Replace data of the initial bank.

Set the flag of the initial bank to 0.

If an N-way TAC scheme, which has N banks

If 'the flag < (N-1)' of the initial bank

Find the highest value of the flag through other banks (say, final bank).

Replace data of the final bank.

Set the flag of the final bank to 0.

For other banks apart from the final bank

Increase the value of the flags by one.

If 'the flag = (N-1)' of the initial bank (say, final bank)

Replace data of the final bank.

Set the flag of the final bank to 0.

For other banks apart from the final bank

If 'the flag < (N-1)'

Increase the value of the flags by one.

Else

Keep the value of the flags.

==========

Figure 21. Pseudo code for the BoPLRU replacement policy

Figure 21 shows the Pseudo code for the BoPLRU. If an N-way TAC scheme employs a n-bit flag, then $N = 2^n$. If n is 1 or 2, it represents a 2-way or 4-way TAC scheme respectively.

For the 2-way TAC scheme, if 'the flag = 0' of the selected bank by the BSL, data in the initial bank will remain while data of the other bank is replaced with new data fetched from memory. After that, the flag of the initial bank will change from 0 to 1. Meanwhile, if 'the flag = 1' for the initial bank, data in the initial bank will be replaced with new data and the flag for the initial bank will change to 0. By doing this, any conflicting data can remain in a bank safely for a while.

For the N-way ($N = 2^n$) TAC scheme in Figure 21, if 'the flag < (N-1)' of the selected bank, it is necessary to find the highest value of the flag for other banks to determine the final bank. After data of the final bank is replaced with new data from memory, the flag will be set to 0 and the value of the other flags except the one of the final bank will be increased by one. Meanwhile, if 'the flag = (N-1)', data for the initial bank will be

replaced by new fetched data and the flag is set to 0. For other banks apart from the final bank, if 'flag < (N-1)', the value of other flags will be increased by one. Otherwise, the value of the flags will be kept since it is the highest value and the flag is not in the final bank but will be in the final bank soon.

The BoPLRU is a kind of modified pseudo-LRU replacement policy that guarantees that recent groups of instructions can be retained in each bank safely.

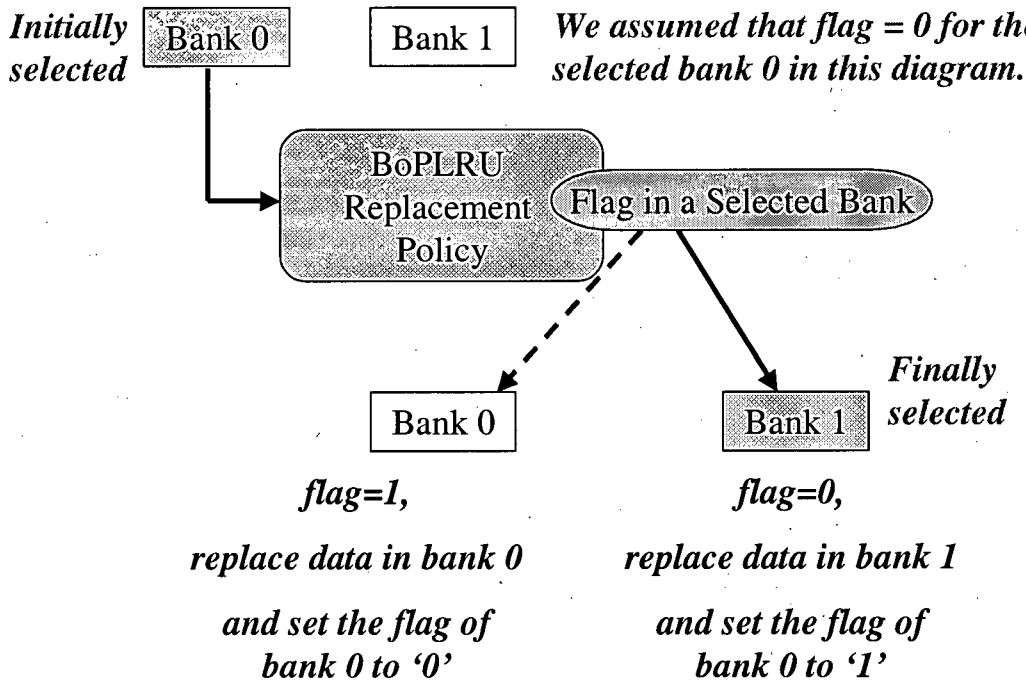


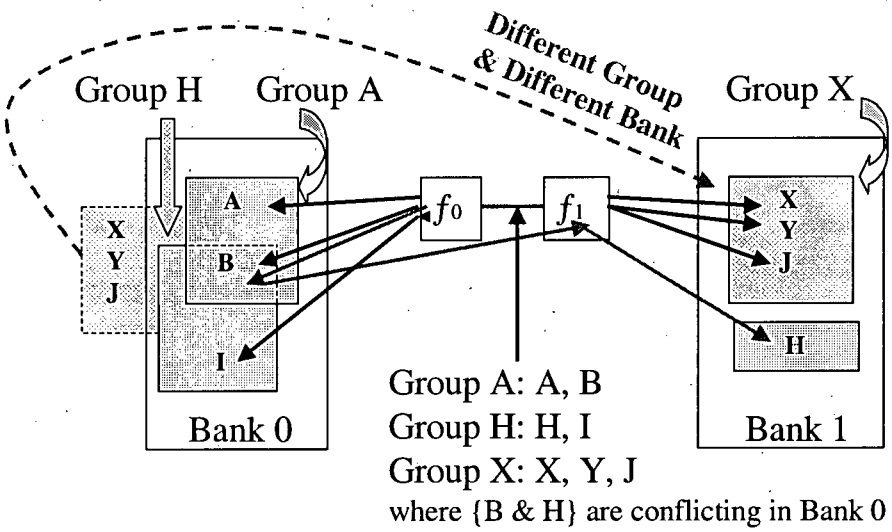
Figure 22. Final bank selection of BoPLRU replacement policy for a 2-way TAC scheme.

As an example, the BoPLRU operation of the 1-bit flag, 2-way TAC scheme, is shown in Figure 22: it is assumed that the BSL initially selects the bank 0 on a cache miss. Therefore, a flag of the selected line in bank 0 is read. If the flag is 1, it is set to 0 and the

data fetched from memory is written into bank 0. Otherwise, the flag is set to '1' and the data is written into bank 1.

2.3.4 Benefit of the TAC scheme

Figure 23 shows how the instructions in Figure 18 are written into each bank (2-way) on a cache miss.



- Guarantees the coexistence of instructions within a group.
- Guarantees the retention of recently used groups of instructions in different banks.

Figure 23. Placement of instructions in a 2-way TAC scheme.

We assume that BSL selects bank 0 for Group A and H, and bank 1 for Group X:

- Instructions A and B of Group A are written into bank 0. It is assumed that the flags for each cache line for Group A are initially set to '0'.

- Instruction H of Group H is written into bank 1 since it conflicts with instruction B of Group A. Therefore, the flag of the cache line for instruction B in bank 0 should be set to '1'.
- Instruction I of Group H is written into bank 0. It is assumed that the flag is initially set to '0'.
- Instructions X, Y, and J of Group X are written into bank 1. It is assumed that the flags of each cache line for Group X are initially set to '0'.

If the instructions in Figure 18 are considered, it can be easily verified that instructions for each group execute in a sequential form. Therefore, the possibility of conflict misses is very low within each group. However, it is reasonable that conflict misses among instructions from different groups can occur easily since the locations of each group of instructions are randomly distributed in the main memory. *Then, how can we effectively reduce the conflict misses among instructions from different groups?*

The answer generally depends on how far the rules of locality in space and/or in time can be satisfied.

1. Locality in Space (Spatial Locality)

   Handy ('93) shows that most computer code is executed repetitively out of a small area. This space is not necessarily in a single address range of the main memory, but may be spread around quite significantly. That is why the principle of spatial locality refers to the fact that a calling routine and the subroutine can exist in two very small areas of memory.

2. Locality in Time (Temporal Locality)

Handy ('93) also notes that the same instruction execute in close sequence with each other, rather than being spread through time. That is, a processor is much more likely to access a memory location which it accessed 10 cycles before than one which it accessed 10,000 cycles before.

The benefit of a TAC scheme comes from satisfying these rules of locality:

- The TAC satisfies spatial locality by grouping instructions according to an effective policy (calling routine and subroutine) and by guaranteeing the co-existence of instructions within a group;
- The TAC satisfies the temporal locality by guaranteeing the retention of recently used groups of instructions in different banks by using the BoPLRU.
- If the frequency of occurrence of procedure call/returns increases, it is expected that the TAC scheme will work even better than other conventional cache schemes.

2.3.5 Examples of cache misses: a 2-way TAC scheme vs. a 2-way skewed-associative

In section 2.2, Gonzalez et al ('97) showed that a 2-way skewed-associative is the most effective cache scheme among the conventional cache schemes such as direct-mapped, 2-way set-associative, 4-way set-associative, hash-rehash, column-associative, victim, and fully-associative schemes. The 2-way skewed-associative scheme can reduce conflict misses most effectively among conventional cache memory schemes.

In this section, cache misses for a 2-way TAC scheme are compared with a 2-way skewed-associative scheme, which is known as the most effective of the conventional cache schemes. It is assumed that:

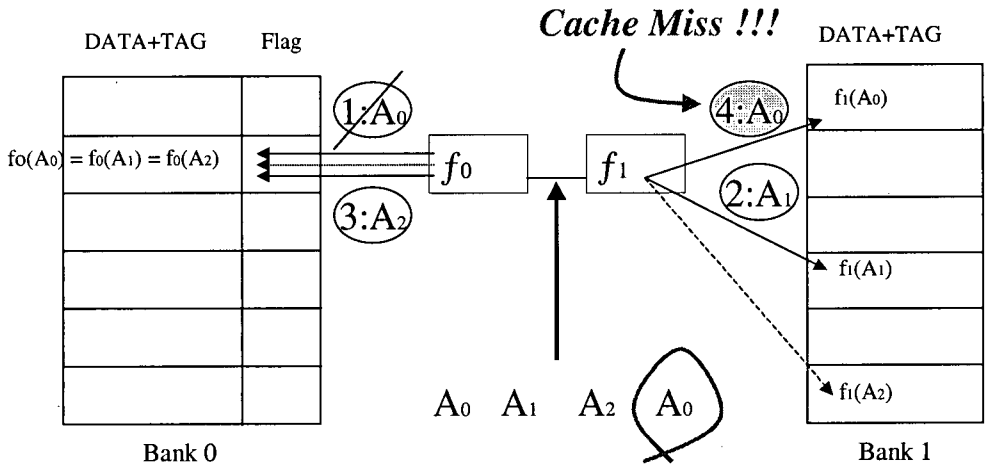
Address a, b, and c compete for the same location in bank 0, but they do not map to the same location in bank 1:

$f_0(a) = f_0(b) = f_0(c)$, $f_1(a) \neq f_1(b) \neq f_1(c)$: Where, f_0 and f_1 are XOR mapping functions.

- The order of fetching addresses: $a \rightarrow b \rightarrow c \rightarrow a$.

An Example for a 2-way skewed-associative

Figure 24 shows cache misses of a 2-way skewed-associative scheme for the above instructions.



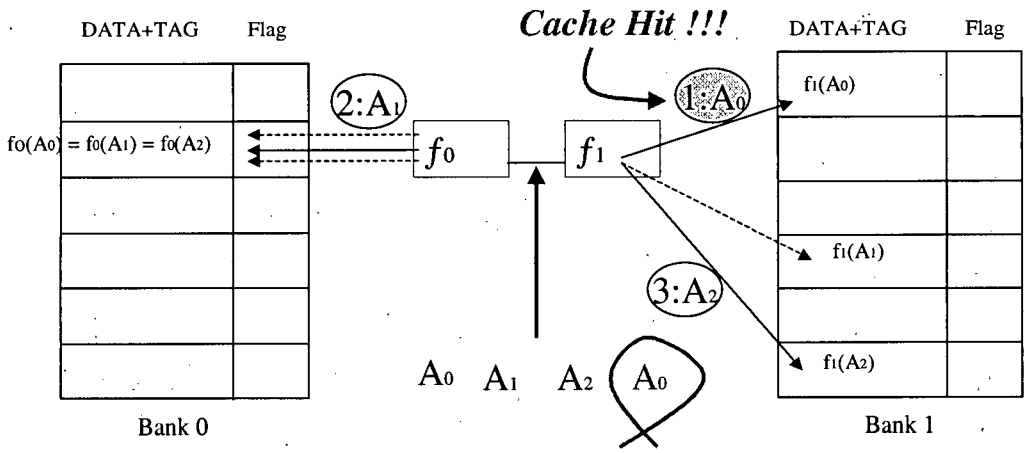
f_0 and f_1: Mapping Function
A_0, A_1, and A_2: Instructions with the same location in Bank 0
Try to avoid ***conflict misses per instruction*** according to the status of the Flag (Flag = 0 → Bank 0 or Flag = 1 → Bank 1)
Flag = 0: initial condition

Figure 24. An example for a 2-way skewed-associative scheme

In Figure 24, there are three initial cache misses for a, b, and c, where a and c are located in bank 0 and b is located in bank 1 according to mapping functions and the flag. Since a and c have the same location in bank 0, a is replaced with c and the flag is set to 1 for the next conflict. Therefore, a can next be located in bank 1. In a 2-way set-associative scheme, the flag is located in cache lines of bank 0 only.

An Example for a 2-way TAC scheme

Figure 25 shows cache misses of a 2-way TAC scheme for the same instructions as the 2-way skewed-associative scheme.



f_0 and f_1: Mapping Function
A_0,A_1, and A_2: Instructions with same location in Bank 0
A_0, A_1, and A_2 → Bank 0 (Flag in Bank 0: 0→1→0→1).
Flag = 0 in Bank 0 → Bank 1 or Flag = 1 in Bank 0 → Bank 0.
All Flags = 0: initial conditions

Figure 25. An example for a 2-way TAC scheme

In Figure 25, there are three initial cache misses for a, b, and c, where a and c are located in bank 1 and b is located in bank 0 according to mapping functions and the flag.

Since there is no conflict miss among a, b, and c between two banks, the last address is where a can be a cache hit. In a 2-way TAC-scheme, each cache line of each bank has its own flag for avoiding conflict misses.

Figure 24 and Figure 25 show that a 2-way TAC scheme works better than a 2-way skewed-associative scheme since the 2-way TAC scheme can reduce conflict misses better than the 2-way skewed-associative scheme.

2.4 Experimental environment

2.4.1 Simulation methodology

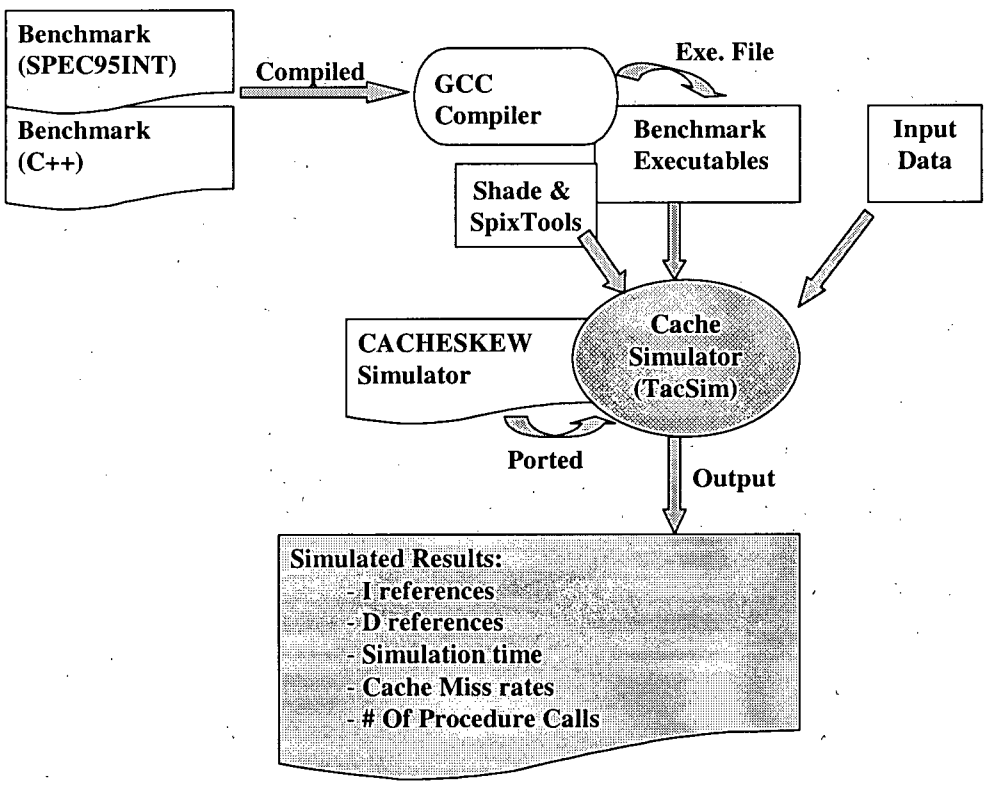


Figure 26. Simulation methodology with benchmark programs and various tools.

Figure 26 shows an overview of our simulation methodology:

- First, SPEC95INT and C++ programs were compiled by using the GNU gcc 2.6.3 and 2.7.2 compiler;
- Second, TACSim (cache simulator) is used to run each executable benchmark with its input data. TACSim was developed by using the Shade, SpixTools, and CAHCESKEW simulator. Shade and SpixTools are tracing and profiling tools developed by Sun Microsystems. Shade executes all the program instructions and passes them on to the cache simulator, TACSim. SpixTools is used for collecting information for static instructions. CACHESKEW is a cache simulator developed by Seznec & Hedouin ('97) that not only simulates most cache schemes such as direct, n-way set-associative and skewed-associative schemes, but also runs several XOR mapping functions and replacement policies such as LRU (Least Recently Used) and Pseudo LRU, etc. The TAC scheme simulator is added into TACSim along with the BoPLRU replacement policy;
- Finally, cache miss rates, the number of instructions and data references, simulation time, etc were collected as outputs.

In Figure 26, *Shade* is a tool that dynamically executes and traces SPARC v9 executables. Using Shade, the trace information desired can be specified. This means that the trace information can be dynamically handled in any manner. It is possible to collect any detailed information for every instruction and opcode dynamically. For example, it is possible to obtain the data for the total number of call instructions, program counter, opcode fields, etc. This information is used for our simulation tool, TACSim.

2.4.2 Benchmarks

Table 3 describes the benchmark programs. Five of the SPEC95 integer programs were used for our simulation – gcc, go, m88ksim, compress, perl, and li. These are the same programs used by Radhakrishnan & John ('98). The next suite of programs is written in C++ and has been used for investigating the behavior between C and C++ [Calder et al '94] [Holzle & Ungar '94]. These programs are deltablue, ixx, and eqn.

Table 4 provides a description of the run-time characteristics of the benchmarks. Dynamic instructions represent the number of instructions executed by each program. It also shows that the number of instructions (function size) per call in the C programs is about two times larger than that of the C++ programs (as a harmonic mean).

Program	Input	Description
SPEC95 CINT: C Programs		
go	2stone9.in	Plays the game Go against itself
gcc	amptjp.i	Compiles pre-processed source
m88ksim	ctl.raw	Simulates the Motorola 88100 processor
compress	test.in	Compresses large text files
perl	scrabble.pl scrabble.in	Performs text and numeric manipulations
li	train.lsp	Lisp interpreter
Suite of C++ Programs		
deltablue	3000	Incremental dataflow constraint solver
ixx	object.h som_plus_fresco. idl	IDL parser generating C++ stubs
eqn	eqn.input.all	Type setting program for mathematical equations

Table 3. Benchmark descriptions

Program	Dynamic instructions	# of procedure calls	Instructions/call
SPEC95 CINT: C Programs			
go	584,163,226	1,610,807	362.65
gcc	250,494,615	5,203,867	48.13
M88ksim	850,957	16,796	50.66
compress	41,765,761	1,355,389	30.81
perl	63,028,127	2,611,048	24.14
li	189,184,575	7,971,176	23.73
Suite of C++ Programs			
deltablue	42,148,983	1,478,007	28.52
ixx	31,829,777	1,404,978	22.65
eqn	58,401,832	1,999,175	29.21
C Mean	4,894,178	97,407	37.67
C++ Mean	41,513,735	1,588,521	26.45

Table 4. Benchmark characteristics

2.5 Experimental results

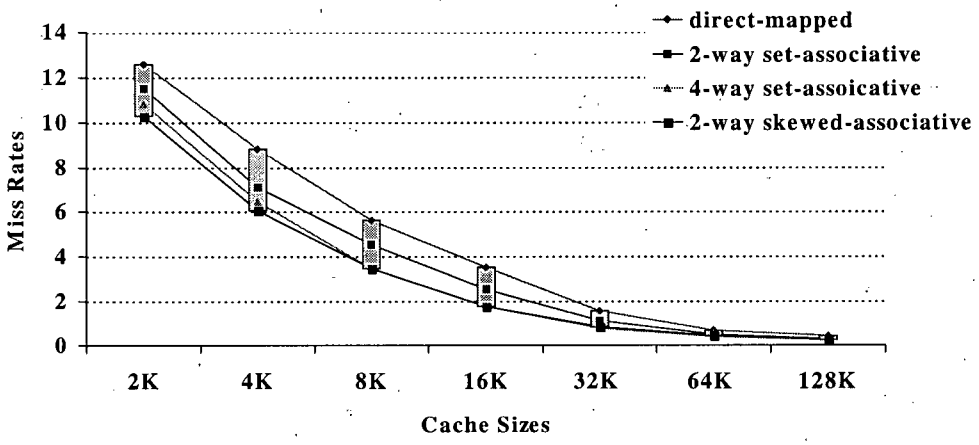
The performance metrics used for comparison of different cache schemes are the instruction cache miss rates and branch misprediction rates. BSL was implemented with a 2-bit counter and the BoPLRU with a 1-bit (2-way) and 2-bit (4-way) flag. If the counter size of the BSL is greater than 4 bits, the instruction cache miss rates are slightly higher than a small-sized counter with less than 2 bits (refer to section 2.5.2). In addition, since Hill & Smith ('89) showed that there is little benefit in increasing cache associativity over

4, experiment results of 2-way and 4-way associativity for the TAC and skewed-associative caches were collected.

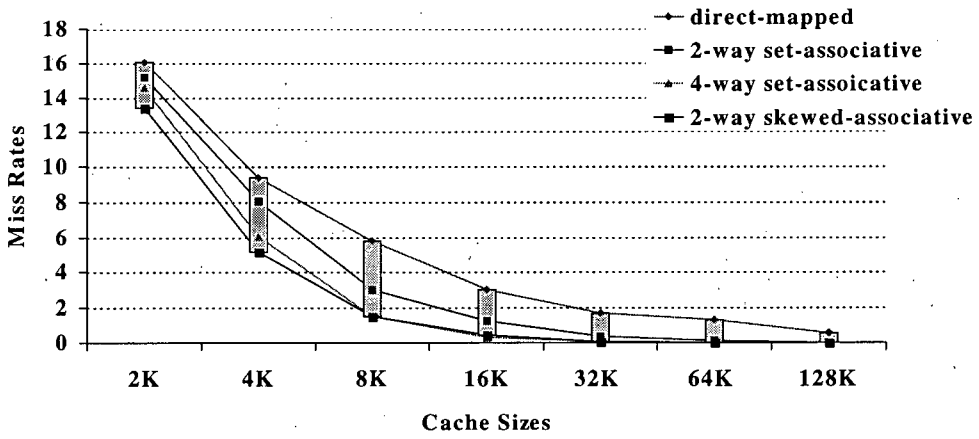
In Table 4, the C benchmark programs, from *go* to *li*, execute from 23 to 48 instructions per call except *go*. The SPECint95 instead of the SPEC2000 were used for our simulation since there have been no experimental results for cache schemes by using the SPEC2000. For "go", since the number of instructions per call is much bigger than other C programs, it will be excluded from all averages in sections 2.5.2 and 2.5.4. "Perl" also is excluded from all averages in sections 2.5.2 and 2.5.4 since it executes 24 instructions per calls like *li* and takes too much time to get a simulation result.

2.5.1 Cache Misses vs. Cache Sizes

Much research has been done to determine the relationship between the cache size and cache miss rates. For our research, 4 cache schemes were simulated with C and C++ benchmark programs in Figure 26: The 4 schemes are direct-mapped, 2-way set-associative, 4-way set-associative, and 2-way skewed-associative; The C programs include go, gcc, m88ksim, li, and compress; The C++ programs are deltablue, ixx, and eqn. The range for the simulated cache sizes is from 2Kbytes to 128 Kbytes according to three different cache line sizes including 8 bytes (Figure 26 (a) and (b)), 16 bytes (Figure 26 (c) and (d)), and 32 bytes (Figure 26 (e) and (f)). The bars in Figure 26 represent the difference between the highest and the lowest miss rates for each cache size. The purpose of the bars is to show which cache sizes could benefit from efficient cache schemes for reducing cache misses.

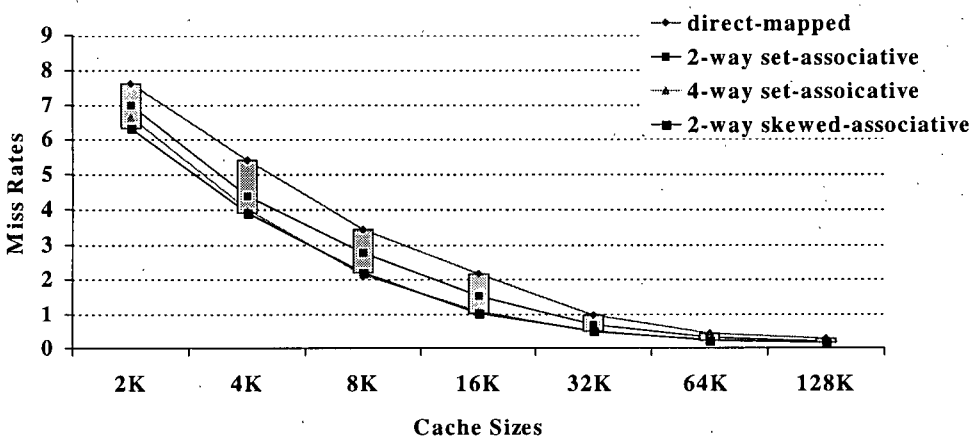


(a) Miss rates vs. Cache sizes for C programs (8 bytes of cache line size)

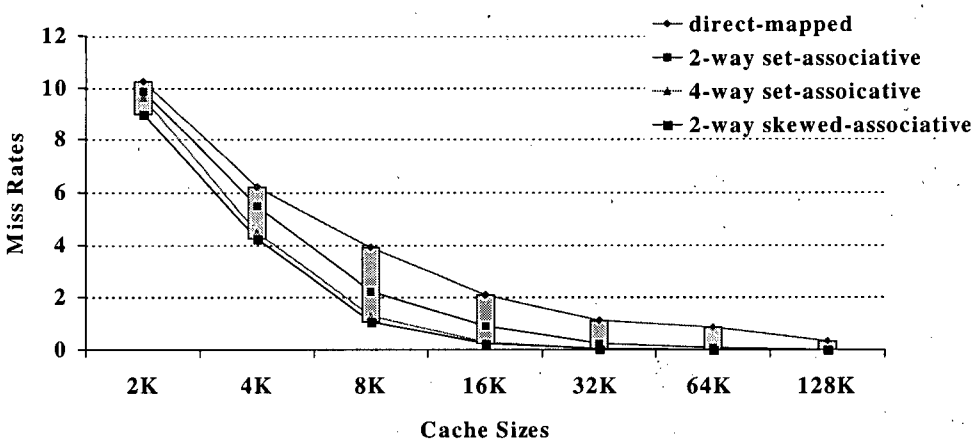


(b) Miss rates vs. Cache sizes for C++ programs (8 bytes of cache line size)

Figure 27. Comparisons for cache misses according to the cache sizes (4 cache schemes).

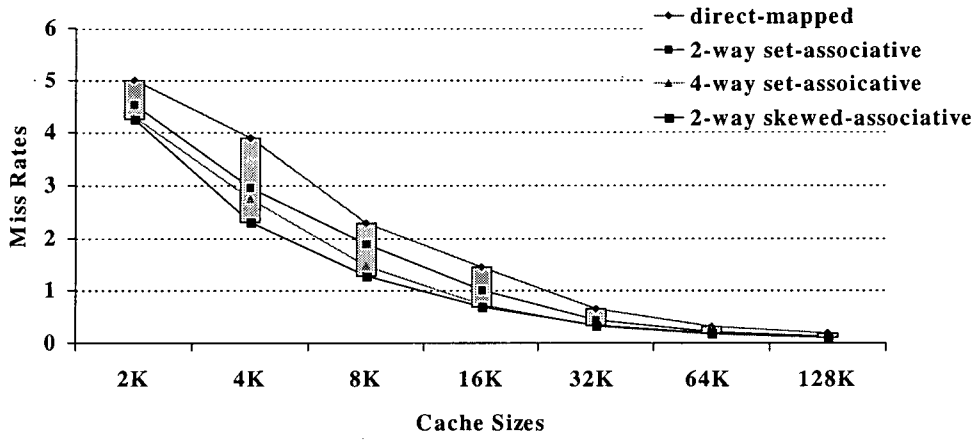


(c) Miss rates vs. Cache sizes for C programs (16 bytes of cache line size)

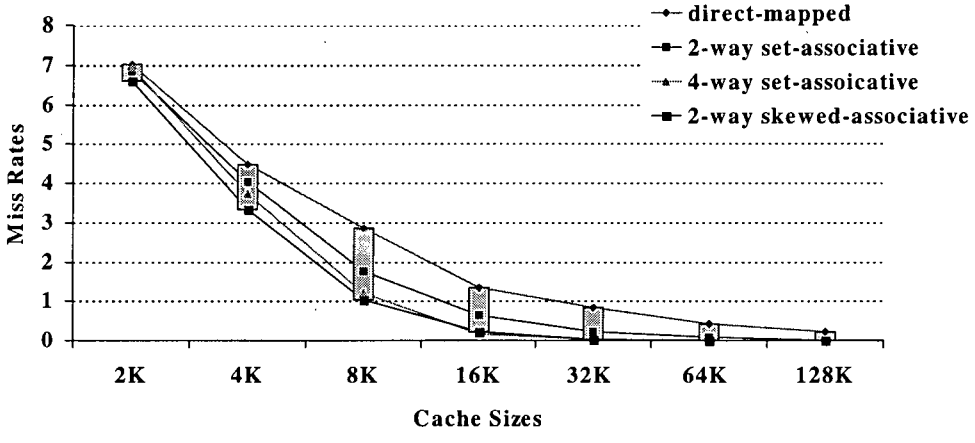


(d) Miss rates vs. Cache sizes for C++ programs (16 bytes of cache line size)

Figure 27. (continued) Comparisons for cache misses according to the cache sizes (4 cache schemes).



(e) Miss rates vs. Cache sizes for C programs (32 bytes of cache line size)



(f) Miss rates vs. Cache sizes for C++ programs (32 bytes of cache line size)

Figure 27. (continued) Comparisons for cache misses according to the cache sizes (4 cache schemes).

Figure 27 (a), (c), and (e) show cache miss rates for the C programs. Meanwhile, Figure 27 (b), (d), and (f) show the results for the C++ programs. Results for the C programs show that if cache sizes are 4 Kbytes to 16 Kbytes, it is useful to have a more efficient cache scheme since cache miss rates can be reduced considerably. In the case of the C++ programs, if cache sizes are 4 Kbytes to 32 Kbytes, a more efficient cache scheme would be useful for reducing cache misses.

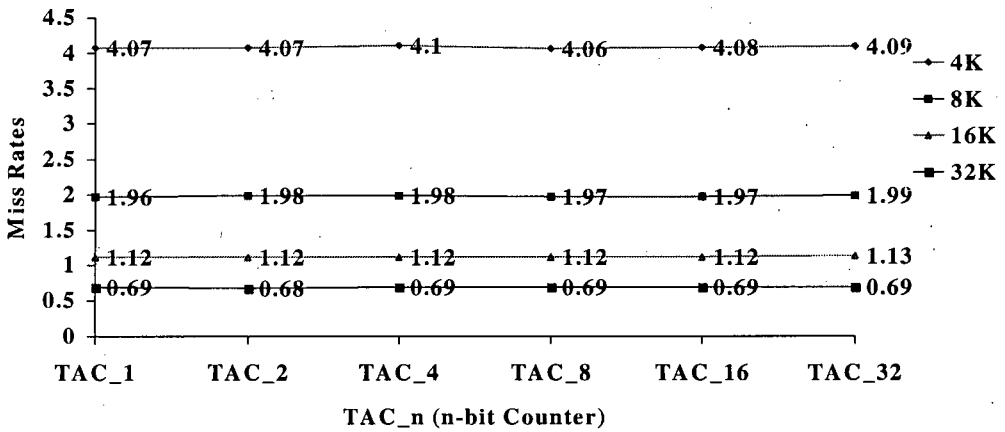
In general, if cache sizes are less than 2 Kbytes or bigger than 32 Kbytes, the cache misses are similar whatever cache scheme is used. Figure 27 tentatively shows that *it is quite reasonable to use a more sophisticated cache scheme for reducing cache misses between 4 Kbytes and 32 Kbytes of cache size.* As microprocessor technology changes, it is widely accepted that small-sized on-chip L1 caches need to replace large external caches.

2.5.2 Bank switching vs. Procedure Calls

In a TAC scheme, the BSL (bank Selection Logic) works to select a bank initially on a cache miss. This section presents the most efficient size of x-bit counter which BSL employs for selecting banks. As we discussed in section 2.5.1, we primarily investigated cache sizes that are less than 32 Kbytes. Various cache sizes of 2-Way TAC scheme were simulated with 7 benchmark programs to determine the most effective x-bit counter size.

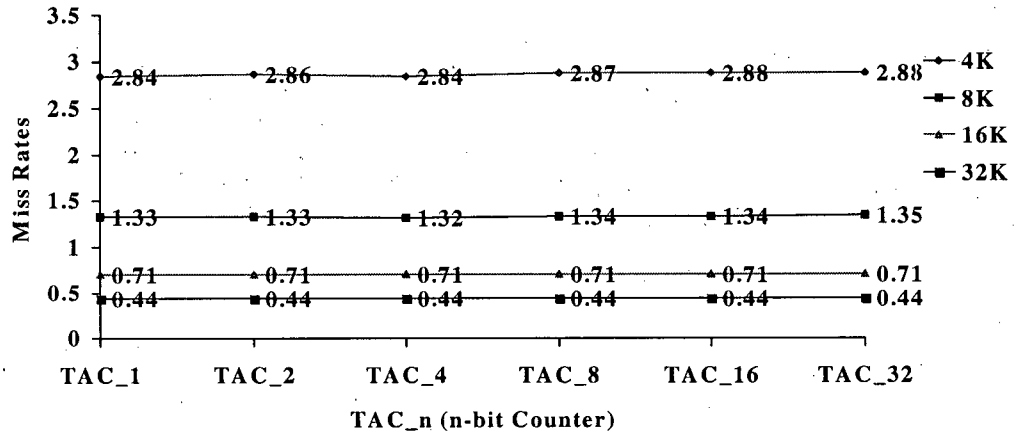
In Figure 28 and Figure 29, TAC_k means that BSL selects a bank for every k call instructions on a cache miss. For example, if k = 2, then every two calls change the bank on a miss. As we discussed in section 2.3, the n-MSBs (Most Significant Bit) of an x-bit

counter represents a bank for the current instruction. Therefore, if k = 2 and n = 1, then a 2-bit counter is needed because {00, 01} → bank 0 and {10, 11) → bank 1. If k = 8 and n = 1, then a 4-bit counter is needed because {0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111} → bank 0 and {1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111} → bank 1. Hence, k = 2^{x-n}, and a x bit counter for the BSL is needed.

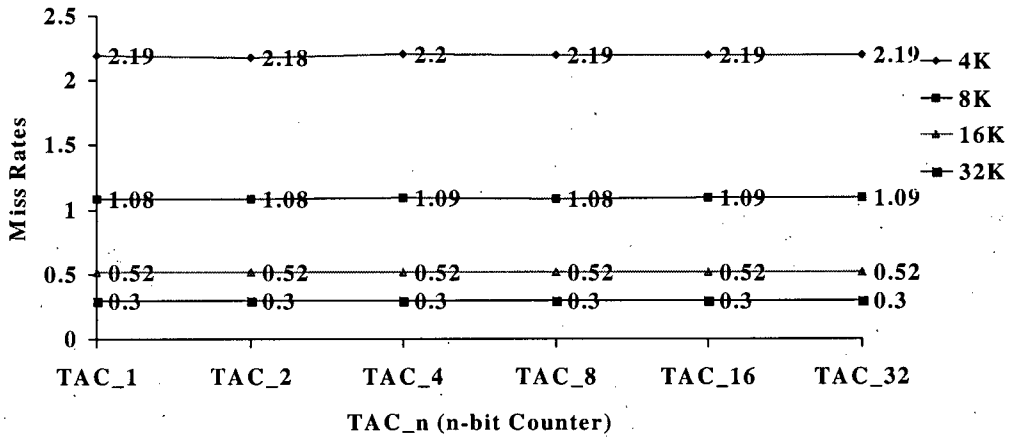


(a) Miss rates vs. TAC_n for C programs (8 bytes of cache line size)

Figure 28. Cache miss rates according to the sizes of the n-bit counter (C programs).



(b) Miss rates vs. TAC_n for C programs (16 bytes of cache line size)



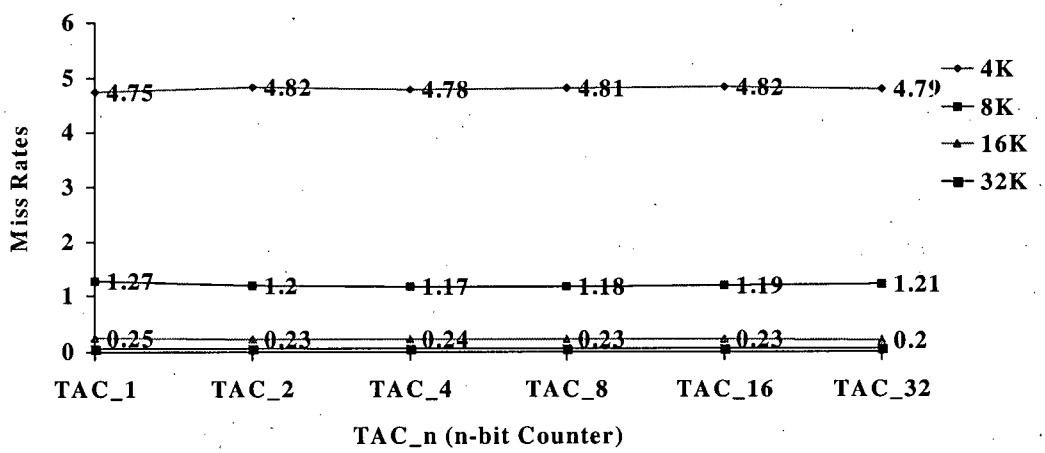
(c) Miss rates vs. TAC_n for C programs (32 bytes of cache line size)

Figure 28. (continued) Cache miss rates according to the sizes of the n-bit counter (C programs).

The following discussion pertains to Figure 28:

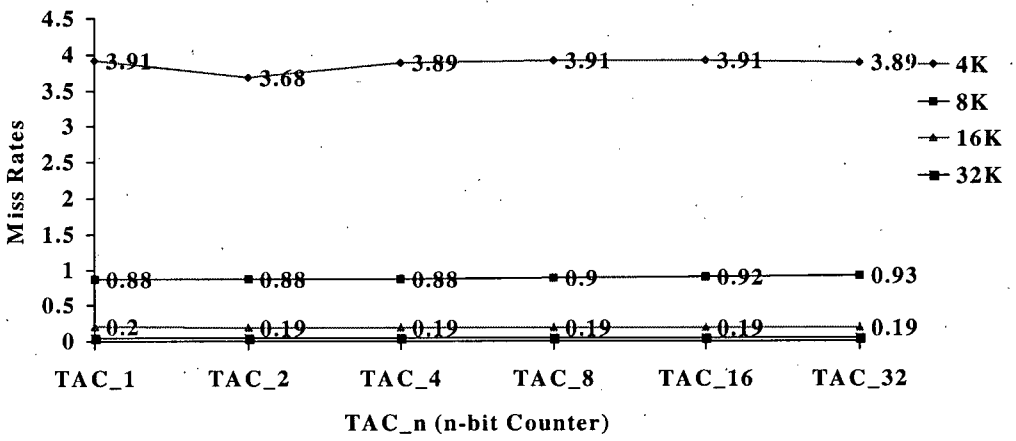
- Four C programs (gcc, m88ksim, li, and compress) were used for determining the most effective x-bit counter for C programs.
- The results of the C programs show that:

  From Figure 28 (a), (b), and (c), if a cache line size is larger (32 bytes), cache miss rates can be slightly reduced by using a smaller x-bit counter (say, less than 4-bit counter).

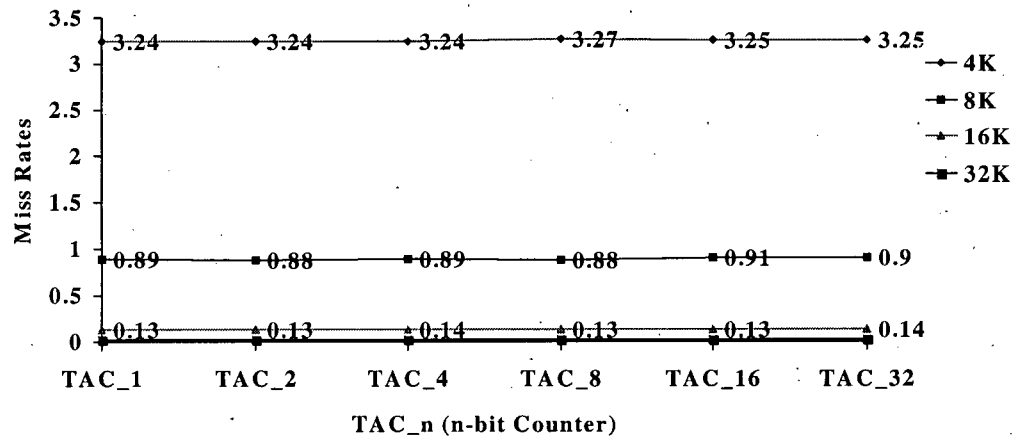


(a) Miss rates vs. TAC_n for C++ programs (8 bytes of cache line size)

Figure 29. Cache Miss rates according to the sizes of the n-bit counter (C++ programs).



(b) Miss rates vs. TAC_n for C++ programs (16 bytes of cache line size)



(c) Miss rates vs. TAC_n for C++ programs (32 bytes of cache line size)

Figure 29. (continued) Cache Miss rates according to the sizes of the n-bit counter (C++ programs).

The following discussion relates to Figure 29:

- 3 C++ programs (deltablue, ixx, and eqn) were used for determining the most effective x-bit counter for object-oriented languages such as C++ and Java.
- The results of the C++ programs show that:

  From Figure 29 (a), (b), and (c), if a cache line size is larger (32 bytes), cache miss rates can be slightly reduced by using a smaller x-bit counter (say, less than 2-bit counter).

In conclusion, *a small-sized counter, less than 2-bit for the case of a 2-way TAC, is recommended for the BSL of a TAC scheme if a cache size is less than 32 Kbytes.*

2.5.3 Instruction Cache Misses for various cache schemes

Results in Table 5 show that the programs in the C++ suite (deltablue, ixx, and eqn) incur higher instruction cache misses than some typical C programs (compress and li). Table 5 also shows that a 2-way TAC scheme removes conflict misses more effectively than a 2-way skewed-associative cache in both C and C++ programs.

Benchmark programs	direct-mapped	2-way	4-way	2-way Skew	2-way TAC	16-way
SPEC95 CINT (C Programs)						
go	6.8691	5.5038	4.8715	5.4535	5.3783	4.9203
gcc	5.9347	5.1155	4.1727	4.1238	3.9645	3.4524
m88ksim	3.8189	2.8224	1.5402	1.3996	1.3202	0.9474
compress	0.0564	0.0475	0.0210	0.0173	0.0163	0.011
li	0.5394	0.4232	0.0834	0.0238	0.0106	0.0052
C++ Programs						
deltablue	3.0746	1.9852	1.3405	1.0326	0.6488	0.2427
ixx	4.7679	2.5423	1.3825	1.1473	0.9444	0.2884
eqn	3.8790	2.0957	1.1382	1.1265	1.0340	0.6186

Table 5. Instruction cache miss rates in percentages (cache size: 8 KB, a line size: 16 bytes)

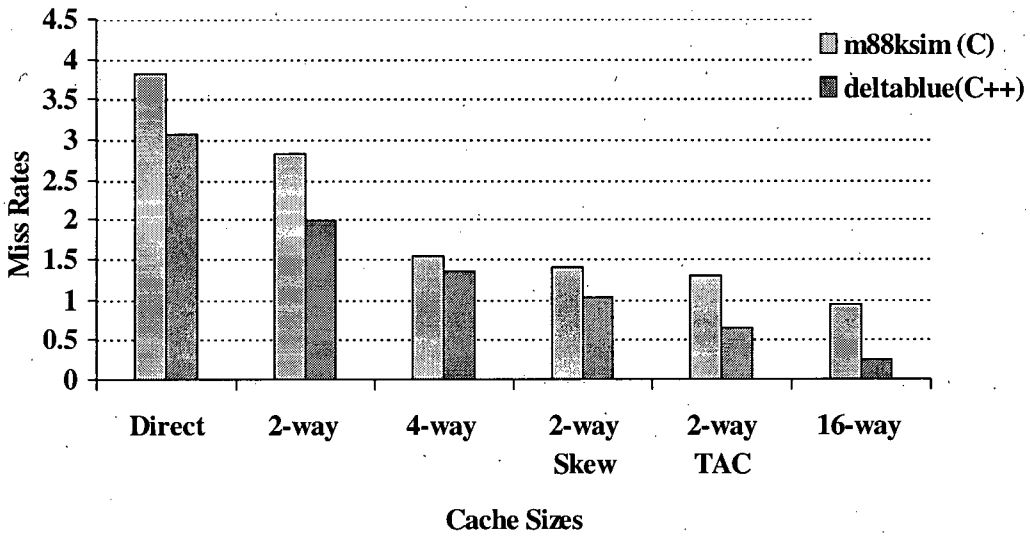


Figure 30. Comparison for instruction cache miss rates between C (m88ksim) and C++ (deltablue) programs (8Kbytes, 16bytes).

Figure 30 shows that the 2-way TAC scheme greatly reduces cache miss rates compared to other cache schemes, with the exception of the 16-way set-associative cache scheme, for both m88ksim (C) and deltablue (C++) programs. The 16-way set-associative cache can be considered a good approximation to a fully-associative cache. In addition, the 2-way TAC scheme for the higher frequency of call instructions (deltablue) works better than that for the lower frequency of call instructions (m88ksim). Thus, the 2-way TAC scheme can replace conventional cache schemes for traditional programs (refer to section 2.5.4) with little or no increase in hardware complexity (refer to section 2.5.6), and is even more suited to object-oriented programs than conventional caches.

2.5.4 Skewed-associative caches vs. TAC schemes

Section 2.5.1 showed that a 2-way skewed-associative scheme can reduce cache misses better than a 2-way or 4-way set-associative scheme. As we discussed in section 2.2, Gonzalez et al ('97) also showed that a 2-way skewed-associative cache offers the lowest miss ratio among several conventional cache schemes and is much lower than a 4-way set-associative cache. This section compares cache miss rates between skewed-associative and TAC schemes. Since there is little benefit in increasing cache associativity over four [Hill and Smith '89], experimental results from 2-way and 4-way associativity for the TAC and skewed-associative caches were collected.

In order to compare cache miss rates between the TAC and skewed-associative caches, we used a formula called IR, Improvement Ratio, such that:

Cache Miss Rates of a 2-way skewed-associative = a;

Cache Miss Rates of a TAC scheme = b;

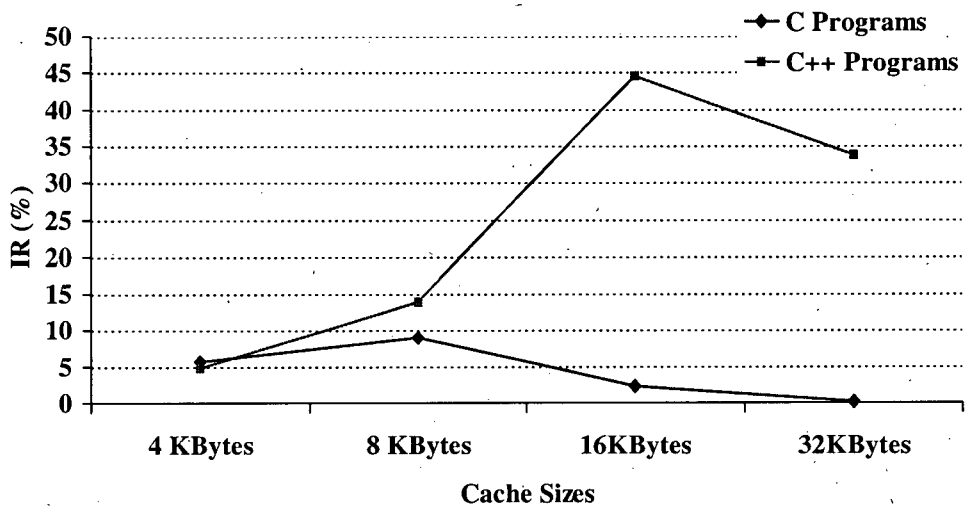
a / b = 1 + n/100 → 'a' has n% more cache miss rates than 'b'.

If n = IR,

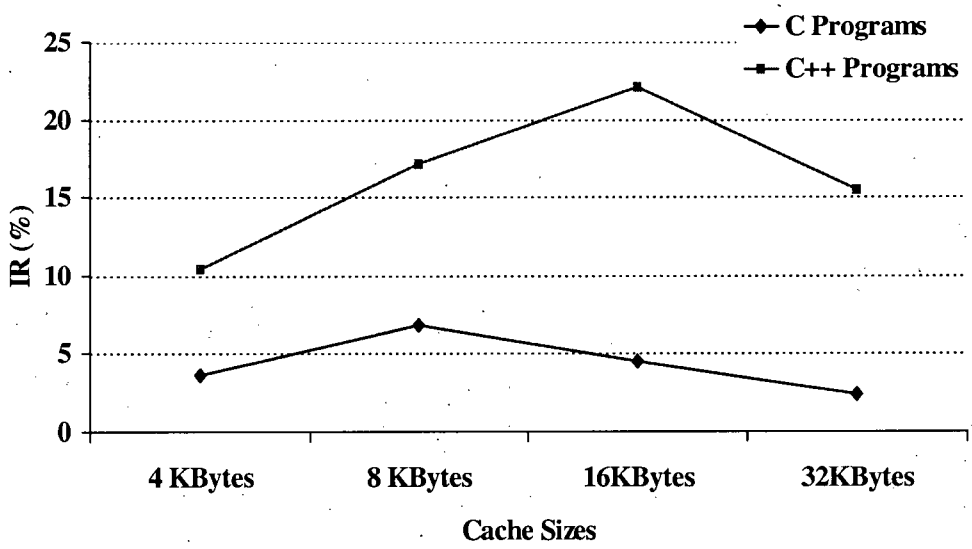
$$IR = ((a - b) / b) * 100\ \% \quad \text{----------} \quad (2)$$

For example, if the cache miss rate of a 2-way skewed-associative scheme is 5%, and that of a TAC scheme is 4%, then, the IR for this case is ((5-4)/4) * 100 = 25%. An IR of 25% means that the 2-way skewed-associative has a cache miss rate of 25% more than the TAC cache.

Therefore, if IR is used for comparing two cache schemes, the improved result can be easily obtained in regard to cache miss rates.

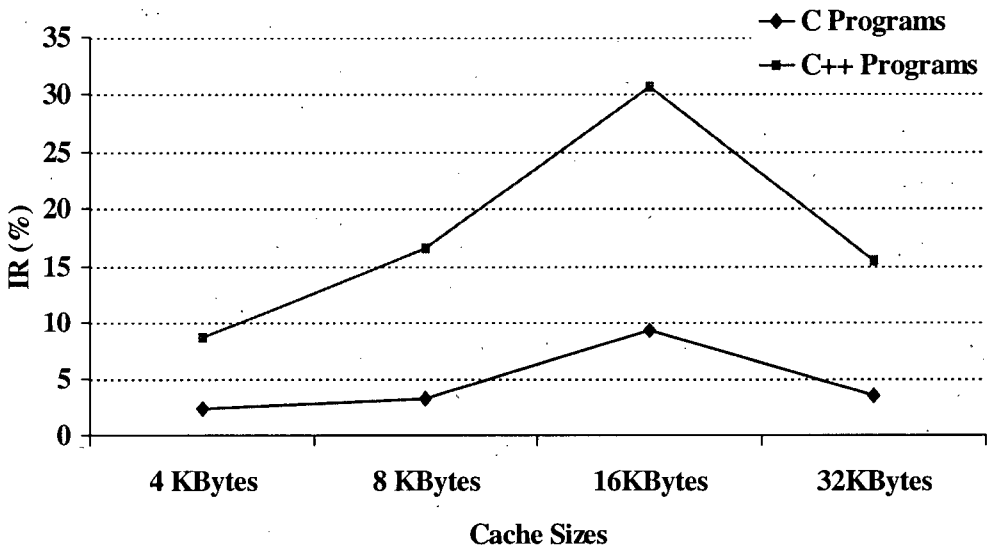


(a) Improvement Ratios for C and C++ Programs (cache line size : 8bytes)



(b) Improvement Ratios for C and C++ Programs (cache line size : 16bytes)

Figure 31. Comparisons for Improvement Ratios between 2-way skewed-associative and 2-way TAC caches.



(c) Improvement Ratios for C and C++ Programs (cache line size : 32bytes)

Figure 31. (continued) Comparisons for Improvement Ratios between 2-way skewed-associative and 2-way TAC caches.

This section shows some graphs with regard to IR between the TAC and skewed-associative caches derived from the tables in Appendix A.

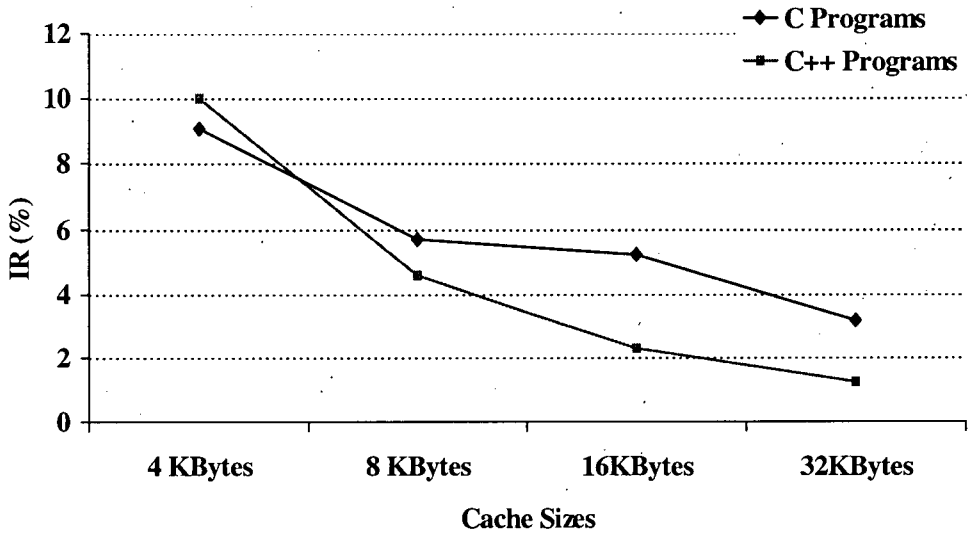
In Figure 31, 4 C programs (gcc, m88ksim, li, and compress) and 3 C++ programs (deltablue, ixx, and eqn) were used for determining IR between 2-way skewed-associative and 2-way TAC schemes.

The results of Figure 31 show that:

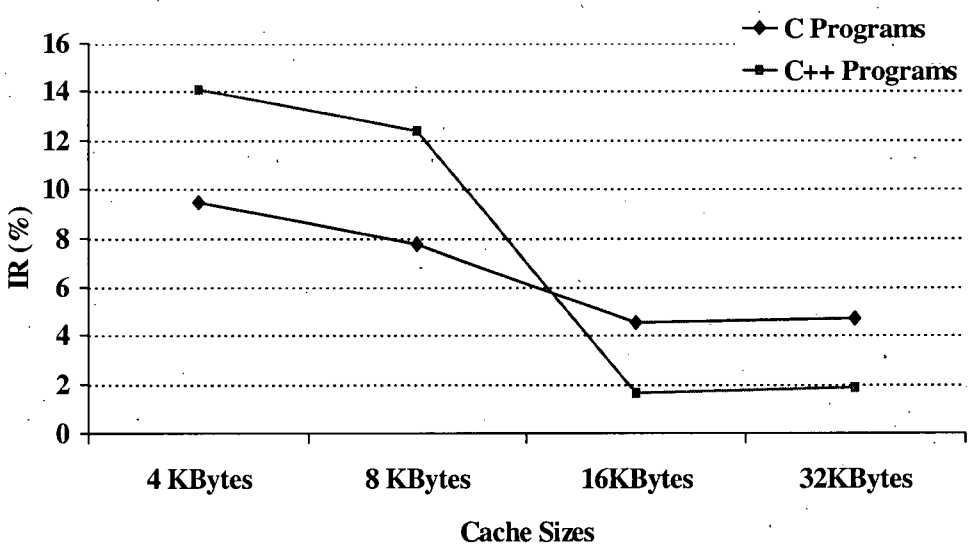
- 2-way TAC schemes can reduce cache misses more effectively than 2-way skewed-associative caches in both C and C++ programs;
- For C programs, the rate of improvement of 2-way TAC schemes over 2-way set-associative schemes range for various cache sizes:
    - From 0.1% (32 Kbytes) to 8.99% (8 Kbytes) for cache line size of 8 bytes;

- From 2.36% (32 Kbytes) to 6.82% (8 Kbytes) for cache line size of 16 bytes;
- From 2.36% (4 Kbytes) to 9.29% (16 Kbytes) for cache line size of 32 bytes;

- For C++ programs, the rate of improvement of 2-way TAC schemes over 2-way set-associative schemes range:
  - From 4.79% (4 Kbytes) to 44.44% (16 Kbytes) for cache line size of 8 bytes;
  - From 10.4% (4 Kbytes) to 22.08% (16 Kbytes) for cache line size of 16 bytes;
  - From 8.66% (4 Kbytes) to 30.71% (16 Kbytes) for cache line size of 32 bytes;
- Therefore, if the cache size is 8 Kbytes (for C programs) or 16 Kbytes (for C++ programs), 2-way TAC schemes can reduce cache misses much better than 2-way skewed-associative caches for all cache line sizes such as 8, 16, and 32 bytes. If cache size is 4 Kbytes or 32 Kbytes, 2-way TAC schemes can reduce cache misses slightly better than 2-way skewed-associative caches for C programs.

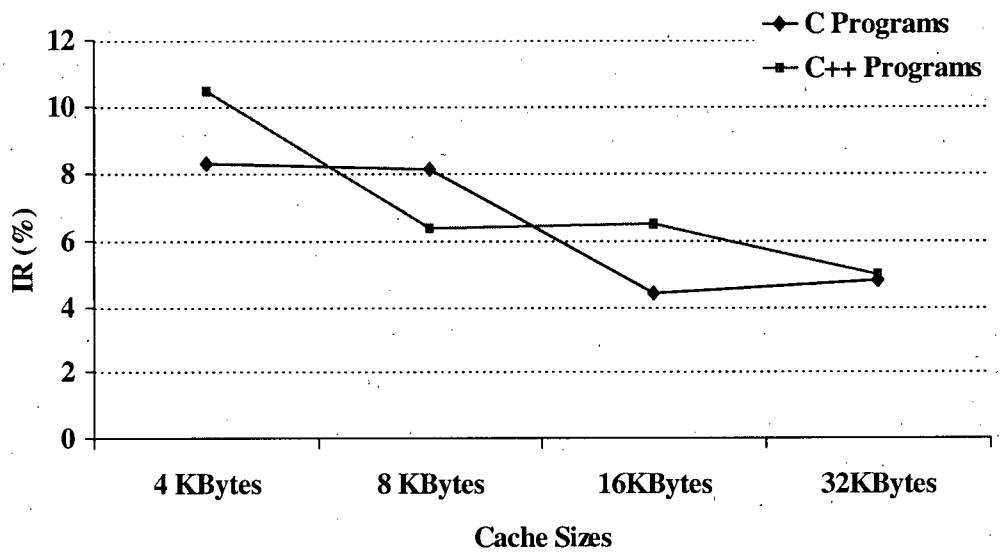


(a) Improvement Ratios for C and C++ Programs (cache line size : 8bytes)



(b) Improvement Ratios for C and C++ Programs (cache line size : 16bytes)

Figure 32. Comparisons for Improvement Ratios between 4-way skewed-associative and 4-way TAC caches.



(c) Improvement Ratios for C and C++ Programs (cache line size : 32bytes)

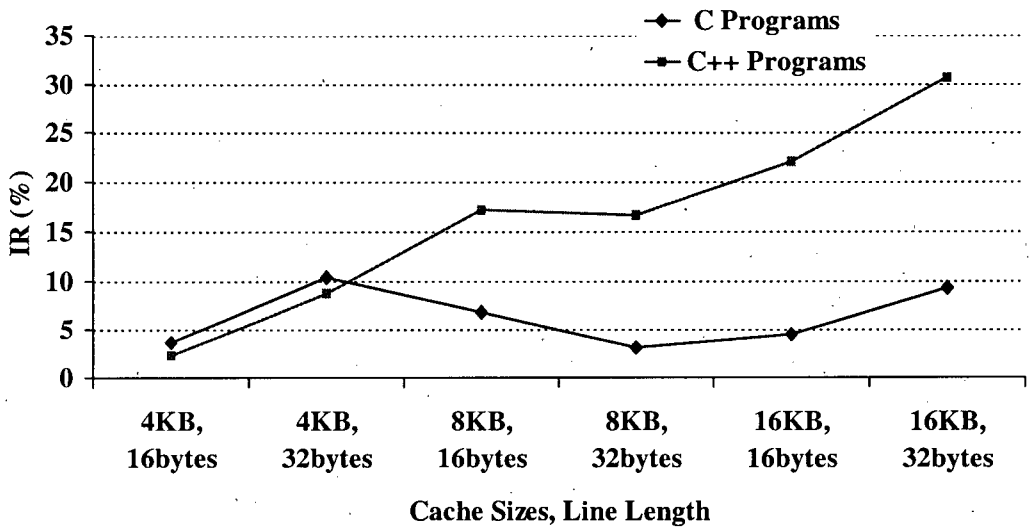
Figure 32. (continued) Comparisons for Improvement Ratios between 4-way skewed-associative and 4-way TAC caches.

In Figure 32, 4 C programs (gcc, m88ksim, li, and compress) and 3 C++ programs (deltablue, ixx, and eqn) were also used for determining IR between 4-way skewed-associative and 4-way TAC schemes.

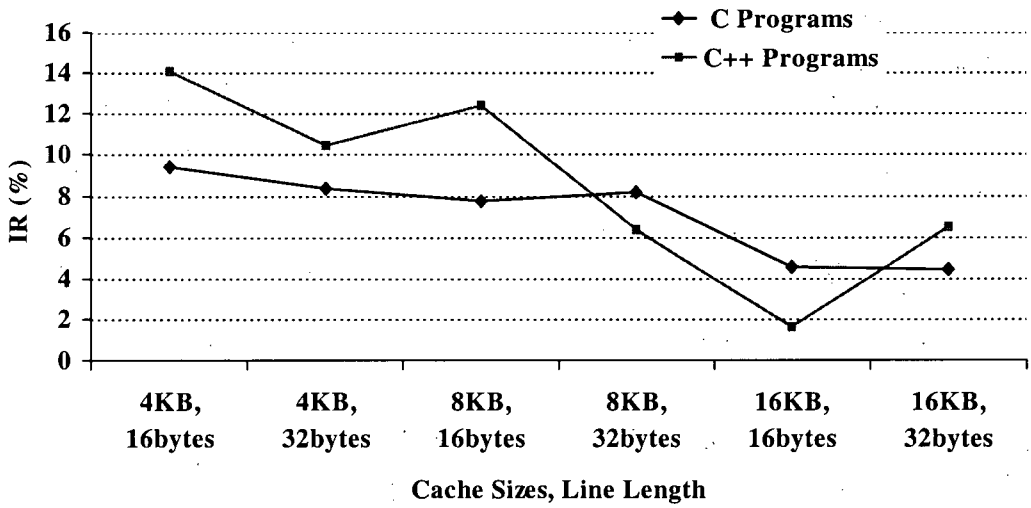
The results of Figure 32 show that:

- 4-way TAC schemes reduce cache misses more effectively than 4-way skewed-associative caches in both C and C++ programs;
- For C programs, the rate of improvement of 4-way TAC schemes over 4-way set-associative schemes range:
    - From 3.2% (32 Kbytes) to 9.06% (4 Kbytes) for cache line size of 8 bytes;
    - From 4.53% (16 Kbytes) to 9.43% (4 Kbytes) for cache line size of 16 bytes;
    - From 4.42% (16 Kbytes) to 8.32% (4 Kbytes) for cache line size of 32 bytes;

- For C++ programs, the rate of improvement of 4-way TAC schemes over 4-way set-associative schemes range:
    - From 1.23% (32 Kbytes) to 10.02% (4 Kbytes) for cache line size of 8 bytes;
    - From 1.63% (16 Kbytes) to 14.09% (4 Kbytes) for cache line size of 16 bytes;
    - From 4.98% (32 Kbytes) to 10.46% (4 Kbytes) for cache line size of 32 bytes;
- Therefore, if the cache size is 4 Kbytes (for C and C++ programs), the 4-way TAC schemes can reduce cache misses much better than 4-way skewed-associative caches for all cache line sizes such as 8, 16, and 32 bytes. If cache sizes are larger than 16 Kbytes, the difference between 4-way TAC and 4-way skewed-associative schemes are reduced since the 4-way TAC or the 4-way skewed-associative caches reduce conflict misses significantly.



(a) Improvement Ratios for C and C++ Programs (2-way, cache line size : 8/16 bytes)



(b) Improvement Ratios for C and C++ Programs (4-way, cache line size : 8/16 bytes)

Figure 33. Comparisons for Improvement Ratios between skewed-associative and TAC caches from 4Kbytes to 8 Kbytes.

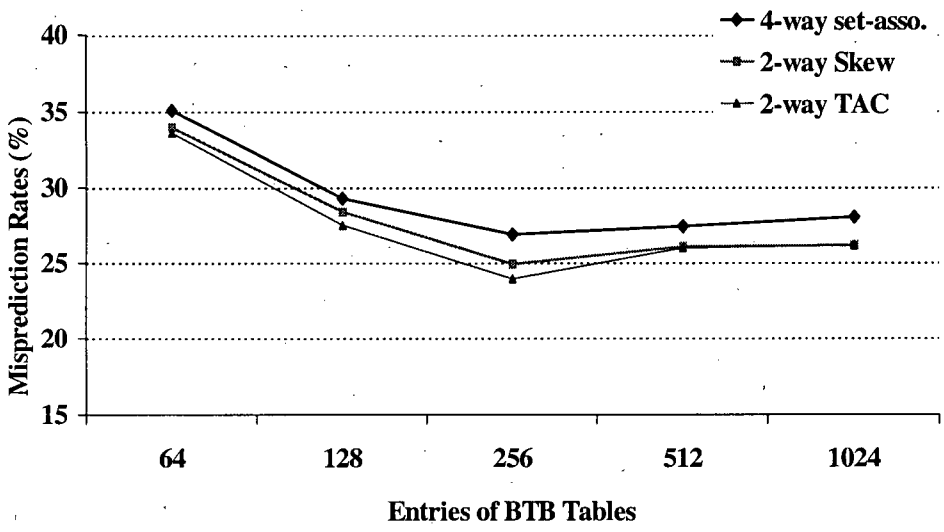
The results of Figure 33 show that:

- (a) 2-way TAC schemes over 2-way skewed-associative caches work well for cache sizes of 8 Kbytes (for C programs) and 16 Kbytes (for C++ programs);
- (b) 4-way TAC schemes work well for the small cache sizes such as 4 Kbytes or 8 Kbytes.

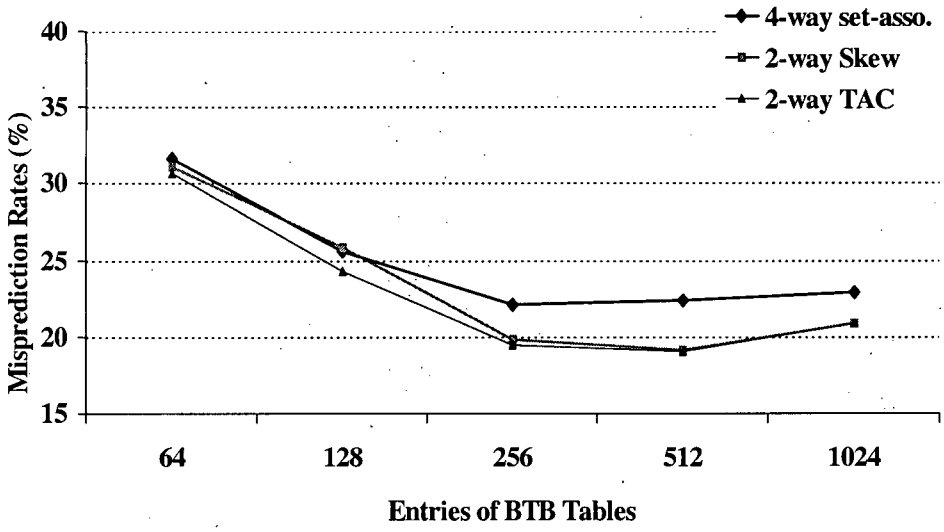
2.5.5 Various cache schemes for the Branch Target Buffer

The Branch Target Buffer (BTB) is a small cache that contains the address of the branch instructions and their target addresses. The BTB is accessed in the fetch stage to predict the state of a branch instruction. If a hit occurs, then the current instruction is a taken branch. The Program Counter (PC) is loaded with the target address from BTB, and fetching starts from the new PC. It has been popular to employ a 4-way set-associative cache for a small-sized BTB table, which has less than 512 entries. Driesen and Holzle ('98) claimed that for a table with 256 entries (64 associativity sets of four) most BTB conflict misses disappear. However, the results of our experiment show that even a BTB with 512 entries (128 associativity sets of four) still suffers from conflict misses.

This section determines the most effective cache scheme for BTB (Branch Target Buffer) among various cache schemes. BTB was simulated with three different cache schemes by using C and C++ benchmark programs in Figure 34. These schemes are 4-way set-associative, 2-way skewed-associative and 2-way TAC scheme. The C programs include go, gcc, m88ksim, li, and perl. The C++ programs are deltablue, ixx, and eqn. The range for the simulated BTB table sizes is from 64 entries to 1024 entries.

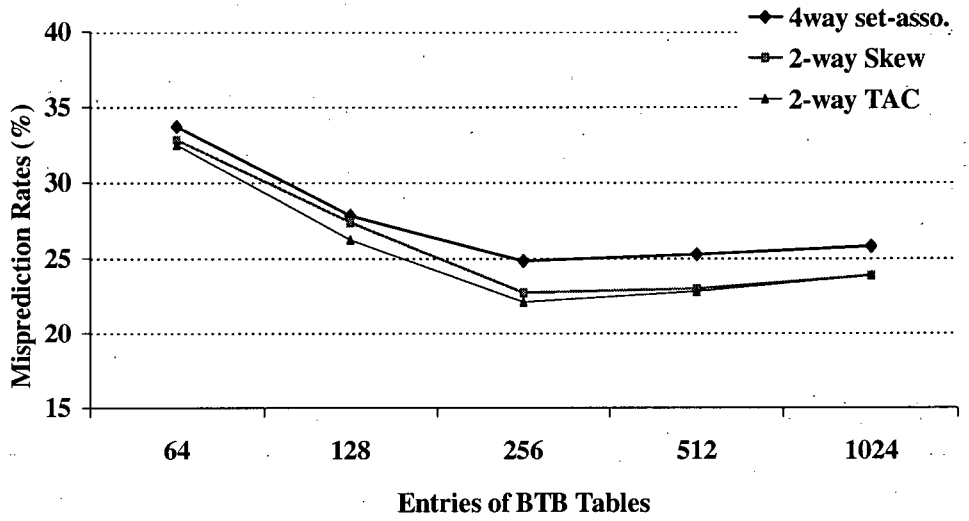


(a) Miss Rates vs. Entries of BTB Tables for C Programs



(b) Miss Rates vs. Entries of BTB Tables for C++ Programs

Figure 34. Comparisons of branch misprediction rates of BTB with a 4-way set-associative, 2-way skewed-associative and 2-way TAC caches.

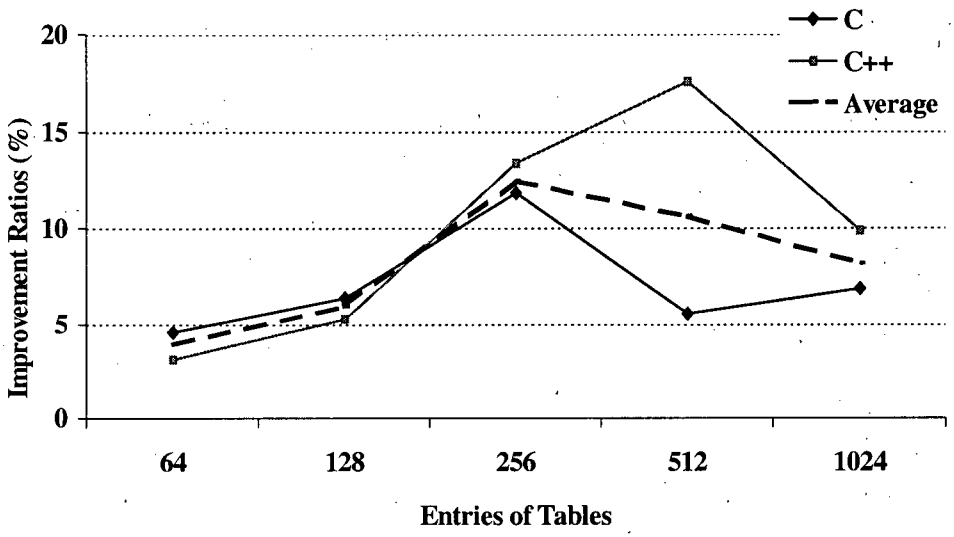


(c) Miss Rates vs. Entries of BTB Tables for C and C++ Programs

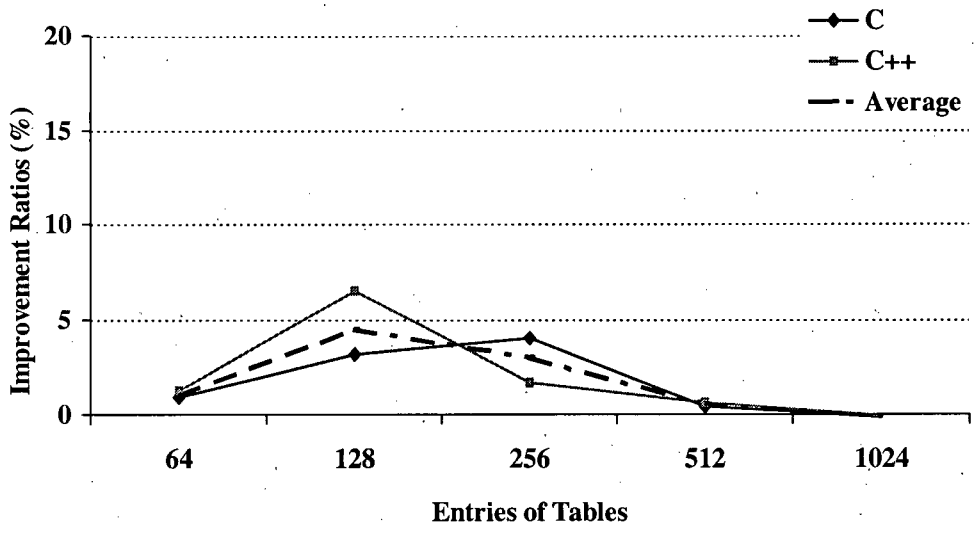
Figure 34. (continued) Comparisons of branch misprediction rates of BTB with a 4-way set-associative, 2-way skewed-associative and 2-way TAC caches.

The results of Figure 34, based data from Appendix A, show that:

- The 2-way skewed-associative and TAC schemes reduce branch misprediction rates more effectively than 4-way set-associative in both C and C++ programs.
- For C programs in Figure 34(a), the 2-way TAC scheme for the 256-entry table of the BTB works better than the other sizes of the BTB table.
- For C++ programs in Figure 34(b), the 2-way TAC scheme for the 512-entry table of the BTB works better than the other sizes of the BTB table.
- The 2-way TAC scheme can reduce branch misprediction rates more effectively for the small-sized BTB tables, i.e., less than 512 entries.



(a) Improvement Ratios between BTB and 2-way TAC Scheme



(b) Improvement Ratios between 2-way Skew and 2-way TAC Scheme

Figure 35. Comparisons for Improvement Ratios among 4-way set-associative, 2-way skewed-associative and 2-way TAC schemes.

In order to compare branch misprediction rates between the 2-way TAC and 4-way set-associative caches, and between the 2-way TAC and 2-way skewed-associative caches, we used a formula called IR, Improvement Ratio, such that:

Branch Misprediction Rates of a 2-way skewed-associative or a 4-way set-associative caches = a;

Branch Misprediction Rates of a 2-way TAC scheme = b;

$a/b = 1 + n/100$. *If* $n = IR$, $IR = ((a - b) / b) * 100\ \%$ ---------- *(3)*

The results in Figure 35(a) show that:

- 2-way TAC schemes work better than 4-way set-associative caches for all table entries, from 64 entries to 1024 entries, in both C and C++ programs.
- For C programs, Improvement Ratios of 2-way TAC schemes over 4-way set-associative caches range from 4% (64 entries) to 11.83% (256 entries).
- For C++ programs, Improvement Ratios of 2-way TAC schemes over 4-way set-associative caches range from 5% (64 entries) to 17.54% (512 entries).
- For all C and C++ programs, Improvement Ratios of 2-way TAC schemes over the 4-way set-associative range from 3.0% (64 entries) to 12.46% (512 entries).

The results in Figure 35(b) show that:

- 2-way TAC schemes work better than 2-way skewed-associative caches for all table entries, which are less than 1024 entries (Figure 35) for both C and C++ programs.
- For C programs, Improvement Ratios of 2-way TAC schemes over 4-way set-associative caches range from 0.39% (512 entries) to 4.0% (256 entries).
- For C++ programs, Improvement Ratios of 2-way TAC schemes over 2-way skewed-associative caches range from 0.63% (512 entries) to 6.50% (128 entries).

- In the case of the 1024 entries for both C and C++ programs, there is no difference between 2-way TAC schemes and 2-way skewed-associative caches.
- For all C and C++ programs, Improvement Ratios of 2-way TAC schemes over 2-way skewed-associative caches range from 0.53% (512 entries) to 4.46% (128 entries) except for the 1024-entry table.

The results in Figure 35 show that 2-way skewed-associative cache and 2-way TAC schemes reduce branch misprediction rates much better than the 4-way set-associative caches. In addition, the 2-way TAC schemes work considerably better than 2-way skewed-associative caches for all table entries, from 64 entries to 1024 entries. However, if a BTB table is greater than 1K entries, our results showed the same results as Driesen and Holzle ('98). Therefore, if the BTB table size is less than 512 entries, the 2-way TAC scheme can be a good solution for reducing branch mispredictions caused by conflict misses.

2.5.6 Comparison for all 2-way schemes

In the previous sections, we discussed that 2-way TAC schemes are the most effective cache schemes to reduce conflict misses for the instruction cache memory or BTB. This section, compares hardware complexity and memory access time among 2-way cache schemes such as 2-way set-associative, 2-way skewed-associative and 2-way TAC schemes.

Contents		2-way set-associative	2-way skewed-associative	2-way TAC
Logic and Indexing	Replacement	LRU, etc.	PLRU, etc.	BSL + BoPLRU
	Indexing	Lower part of address	XOR mapping	XOR mapping or Polynomial, etc.
Hardware	Banks	2	2	2
	Flag	Y (Bank 0)	Y (Bank 0)	Y (Each Bank)
	Bank design	Classical design	Classical design + XOR gates (mapping)	Classical design + XOR gates (map)
	Counter	N	N	Y (BSL)
Access Time		Same	Same (slightly increased by XOR gates)	Same (slightly increased by XOR gates)
Hardware Complexity		Same	Almost Same (Only several XOR gates are added to the 2-way set-associative)	Almost Same (Only a counter is added to the 2-way skewed-associative)
Cache Miss Ratio		High	Medium	Low

1. BoPLRU: Bank-originated Pseudo Least Recently Used, BSL: Bank Selection Logic
2. '**Access Time**' and '**Hardware Complexity**' from 'A. Seznec, A case for two-way skewed associative Caches, Proc. Of the 20th ISCA, May 1993, pp169-178'.

Table 6. Comparison of hardware complexity and access time among three representative 2-way schemes: 2-way set-associative, 2-way skewed-associative, and 2-way TAC schemes.

Table 6 shows various characteristics for three different 2-way schemes:

- The 2-way TAC scheme employs a flag for each bank while other 2-way schemes employ only one bank. However, the flag size will not be a critical factor in increasing the hardware complexity in a 2-way TAC scheme since the flag size is only 1 bit.
- Memory access time of 2-way skewed-associative and 2-way TAC schemes is greater than for 2-way set-associative caches as a result of using XOR mapping functions. However, according to Seznec ('93), the memory access time caused by the XOR

mapping functions will be slightly increased by one xor-gate delay time since several xor gates work in parallel for the XOR mapping functions.

According to Table 6, the 2-way TAC scheme is seen to be a good solution for reducing conflict misses for instruction cache memory and BTB with similar hardware complexity and memory access time compared to 2-way set-associative caches.

2.6 Chapter conclusions

Unlike traditional application programs, object-oriented languages use many small functions during run-time and this is the main factor for conflict misses. This paper presents a new cache scheme called TAC (Thrashing-Avoidance Cache), which effectively reduces instruction cache misses caused by frequent procedural call/returns.

Among conventional cache schemes, the skewed-associative cache offers the lowest miss ratio, which is significantly lower than a 4-way set-associative cache. However, a skewed-associative cache has a limitation in handling conflict misses in object-oriented programs due to the problem of accessing the large number of small functions. The main reason for this is that a skewed-associative cache is designed to reduce conflict misses for individual instructions only. The TAC scheme works not only for individual instructions but also for a group of instructions such as a calling routine and its associated subroutine.

Our simulation results show that:

- TAC schemes (on L1 cache) can improve instruction cache miss rates by up to 9.29% for C programs and 44.44% for C++ programs over skewed-associative caches.
- TAC schemes (on BTB, 2-way) reduce branch misprediction rates more effectively than 4-way set-associative by up to 11.83% for C programs and 17.54% for C++ programs.
- TAC schemes (on BTB, 2-way) also reduce branch misprediction rates better than skewed-associative (2-way) caches by up to 4% for C programs and 6.5% for C++ programs.
- Hardware cost and memory access time in an N-way TAC scheme are similar to a n-way set-associative cache since an N-way TAC scheme employs N banks (N-way) and XOR mapping functions with simple hardware complexity.
- TAC schemes employ an efficient replacement policy. The BoPLRU effectively reduces conflict misses caused by the procedure call/returns by ensuring that the recent groups of instructions are retained in each bank safely.

Future work involves combining TAC schemes with more efficient mapping functions, more effective replacement policies, etc.

Chapter 3 Reduction of Indirect Branch Mispredictions

This chapter presents a new hybrid branch predictor called the GoStay2, which can effectively reduce misprediction rates for indirect branches. The GoStay2 has two mechanisms that are different from other 2-stage hybrid predictors that use a Branch Target Buffer (BTB) as the first stage predictor. First, to reduce conflict misses in the first stage, a new effective 2-way cache scheme is used instead of a 4-way set-associative. Second, to reduce mispredictions caused by an inefficient predict and update rule, a new selection mechanism and update rule are proposed. A simulation program has been developed by using Shade and Spixtools, provided by SUN Microsystems, on an Ultra SPARC/10 processor. Our results show good improvement with these mechanisms compared to other hybrid predictors. For example, the GoStay2 improves indirect misprediction rates of a 64-entry to 4K-entry BTB (with a 512- or 1K-entry PHT) by 14.9% to 21.53% compared to the Cascaded predictor (with leaky filter).

3.1 Introduction

For high performance computer architectures, branch prediction is a key mechanism in achieving high performance on multi-instruction issues. Branches transfer control flow of programs. The next instruction can only be decided after the current instruction is executed. Therefore, if there is no branch prediction scheme, the pipeline always stalls for at least three clock cycles (decode, issue, and execute stages) whenever it meets a branch instruction. A poor branch prediction scheme likewise results in many such stalls, whereas

a good branch prediction scheme reduces stalls. Thus, more accurate branch predictors are desired for reducing the impact on overall system performance.

According to Chang et al. ('97), branches can be categorized as conditional or unconditional, as well as direct or indirect, resulting in four classes. Of the four classes, prediction of conditional indirect branches is typically not implemented [Kalamatianos & Kaeli '98].

Conditional or unconditional direct branch instructions include a target address as part of the instruction. However, unconditional indirect branch instructions obtain a target address indirectly through a register or a pointer variable. Therefore, while direct branch instructions have a single target, indirect ones have multi-targets. Single-target direct branches can be predicted with reported hit-ratios of up to 97% [Yeh & Patt '93]. In contrast, indirect branches with multi-targets are harder to predict accurately. Indirect branches occur frequently in some widely used benchmark programs like SPECint95, and even more frequently in object-oriented languages. The sources of indirect branches are switch statements, virtual function calls, or indirect function calls [Kalamatianos & Kaeli '98][Driesen & Hölzle '98B]. Calder et al. ('94A) investigated behavioral differences between C and C++ programs; C++ programs execute many fewer conditional branch instructions (61.6% vs. 80.0%) and more procedure calls (11.2% vs. 6.3%), indirect procedure calls (3.9% vs. 0.3%), and return instructions (15.1% vs. 6.6%). The above results indicate that handling indirect calls, procedure calls, and returns properly should be important for C++ programs [Calder et al. '94A]. Chang et al. ('97) also showed that indirect branches occur frequently in C++ (object-oriented languages), which are rapidly increasing in popularity.

Conventional branch predictors predict branch direction and generate the target address associated with that direction. BTB-based prediction schemes are the only predictor for indirect branch prediction in conventional branch schemes since an indirect branch needs a full target address instead of direction (taken or not-taken). However, they perform poorly, with a 66% to 76% misprediction rate for indirect branches since the target of an indirect branch can change with every dynamic instance of that branch [Chang et al. '97]. Chang et al. ('97) showed that the small proportion of indirect branches (2 to 3%) for SPECint95 benchmarks could be a critical factor in degrading system performance. Thus, an accurate indirect branch predictor is needed for widely used object-oriented languages such as C++ programs since their indirect branch ratio is at least two to three times higher than that of SPEC benchmarks (C programs) [Chang et al. '97][Calder et al. '94A].

This chapter presents a 2-stage hybrid predictor called the GoStay2, which employs a new cache scheme for the first stage and a new selection mechanism and update rule using a 2-bit flag. The flag is a similar mechanism to the meta-predictor used by McFarling ('93). However, our flag is updated according to the update rule and execution results while the meta-predictor is affected by the execution results only. This chapter shows that the GoStay2 outperforms other 2-stage hybrid predictors such as the Target Cache [Chang et al. '97] and Cascaded predictor [Driesen & Hölzle '98B] by improving the accuracy of indirect branch predictions.

This chapter is organized as follows: Section 3.2 explains related work; section 3.3 presents the new branch architecture with the two mechanisms for reducing indirect

mispredictions; section 3.4 describes simulation methodology and benchmark programs; section 3.5 presents our simulation results; and section 3.6 provides our conclusions.

3.2 Related work

Various branch prediction strategies for indirect branches have been previously studied to improve prediction accuracy. These strategies can be categorized into three main areas:

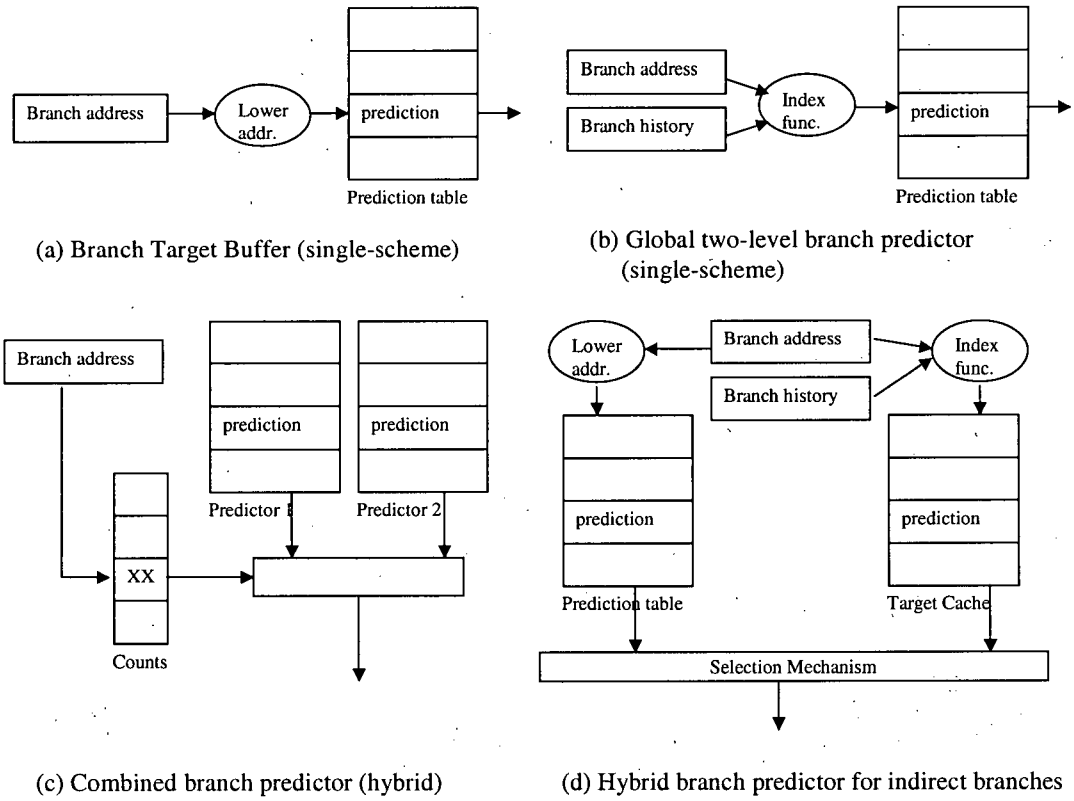
- Indexing functions for accessing predictor tables;
- Selection mechanisms for choosing accurate prediction in hybrid predictors;
- Update rules after resolving a branch.

For each single-scheme predictor, the accuracy of the branch prediction depends on the indexing functions. Most of the research on branch prediction has been done on developing efficient indexing functions.

A hybrid branch predictor combines two or more single-scheme predictors. The performance of the hybrid predictor depends on both indexing functions of each single-scheme predictor and a selection mechanism for a particular predictor. Recently, several selection mechanisms have been proposed to predict indirect branches by using a sophisticated form of the update rules instead of just simple n-bit counters [Chang et al. '97] [Driesen & Hölzle '98B].

3.2.1 Indexing functions for indirect branch predictors

There are two types of branch predictors classified according to the number of component predictors: A single-scheme predictor that has only one predictor and a hybrid predictor that combines two or more single-scheme predictors.



(a) Branch Target Buffer (single-scheme)

(b) Global two-level branch predictor (single-scheme)

(c) Combined branch predictor (hybrid)

(d) Hybrid branch predictor for indirect branches

Figure 36. Various indirect branch predictors

In Figure 36, (a) and (b) represent typical single-scheme predictors. The Branch Target Buffer (BTB) stores both the branch address and target address. If a current branch is found in the BTB, it is predicted as 'taken' with the target address. If there is a misprediction or a first-miss, the branch and target addresses are updated after the execution. When a branch address is not found in the prediction table, it is recognized as a first-miss. In general, a low-order branch address is used as the indexing function to

access the physical line of the BTB. As we discussed before, the BTB-based prediction schemes should not used for indirect branches because of poor prediction accuracy [Chang et al. '97]. For improvement of the BTB, a 2-bit strategy was proposed by Calder and Grunwald ('94). This strategy used a 2-bit counter for limiting the update of the target address in the BTB only after two consecutive mispredictions have occurred. The 2-bit strategy can reduce a misprediction ratio for C++ applications without changing predictions too rapidly. However, the 2-bit strategy is not very successful in predicting the targets of indirect branches in C programs such as SPECint95 benchmarks [Chang et al. '97].

For indirect branches in Figure 36, each single-scheme predictor should hold a target address per cache line instead of just a direction (taken/not taken). The single-scheme predictor in Figure 36(b) shows an indexing function obtained by varying a two-level adaptive scheme described in [Yeh & Patt '93]. This is called a gshare scheme that was introduced by McFarling ('93). The gshare scheme performs better than a two-level [Yeh & Patt '93] predictor by XORing (exclusive oring) the global branch history with the lower bits of a branch address to generate the index into the prediction table. The gshare is considered as one of the highest performance predictors and the best single scheme at all levels of cost [Chang et al. '97][Driesen & Hölzle '98B].

In Figure 36(c), McFarling ('93) introduced the concept of a hybrid branch predictor by combining single-scheme predictors. The combined branch predictor consists of two predictors and a table of 2-bit saturating counters. This counter array is called a meta-predictor and is used to select the more accurate predictor for a current branch. After

resolving a branch, both component predictors are updated, and the meta-predictor is updated to reflect the relative accuracy of the two predictors.

Figure 36(d) shows an indirect hybrid branch predictor that consists of two predictors such as the BTB (Figure 36(a)) and gshare-like single-scheme predictors (Figure 36(b)). This chapter considers only hybrid branch predictors consisting of two single-scheme predictors. Moreover, one of the predictors is a BTB since this chapter compares strategies for simple and effective predictors such as the Target Cache and Cascaded predictor:

- Target Cache - Chang et al. ('97) proposed a predictor by using the Target Cache to improve the accuracy of indirect branch predictions. The Target Cache is similar to the Pattern History Table (PHT) of a 2-level branch predictor except that the Target Cache records the branch target while the PHT holds only branch directions such as taken/not taken. This predictor XORs pattern- or path-based history bits with the branch address to index the prediction table shown in Figure 36(d). The Target Cache can reduce the misprediction rates of indirect branches significantly. For example, a 512-entry Target Cache achieved a misprediction rate of 30.4% and 30.9% for gcc and perl, while a 1K–entry 4-way set-associative achieves rates of 60% and 70.4% [Chang et al. '97];
- Leaky or Strict Filters of the 2-stage Cascaded Predictor - Driesen and Hölzle ('98B) introduced two variants of the Cascaded Predictor, which has two stages; a BTB for the first stage and a gshare-like two-level predictor as the second stage. The small-sized BTB works as a filter and the second stage predictor stores indirect branches that need branch history-based prediction. The second stage uses an indexing function

similar to the Target Cache, such as a path-based branch history XORing with a low-order branch address to index the prediction table shown in Figure 36(d). Driesen and Hölzle ('98B) showed that the two filters (leaky or strict) have slightly different update rules (Table 7). The filtering effect decreased the misprediction rate compared to a non-filtering Cascaded Predictor. For example, a 32-entry BTB filter (first stage) improved the misprediction rate of a 256-entry monopredictor (non filtering) from 11.7% to 10.7% [Driesen & Hölzle '98B].

Kalamatianos and Kaeli ('98) showed that the leaky filter of a Cascaded Predictor improved indirect branch prediction accuracy over the Target Cache in most SPECint95 and C++ benchmarks.

3.2.2 Selection mechanisms and update rules for hybrid predictors

In the combined predictor of McFarling ('93), there are two single-scheme predictors such as p1 and p2. The meta-predictor, a table of 2-bit counters, is used to select one of two predictors as a selection mechanism. A 2-bit counter reflects the states "strongly p2 (11)", "weakly p2 (10)", "weakly p1 (01)", and "strongly p1 (00)". For example, when a branch is predicted, each single-scheme predictor is queried. If the counter is '00', then p1 is selected for the branch prediction. When the branch commits, both predictors are updated and the meta-predictor is updated to favor the predictor that had the correct prediction [McFarling '93] [Grunwald et al. '98].

While conventional hybrid branch predictors use a meta-predictor as a selection mechanism, the Target Cache and Cascaded predictor have no such meta-predictor.

Instead, selection is done by predict rule; both stages are examined for finding a current branch address. If both stages have the current branch, the second stage predictor takes precedence. If not, the target address in any stage, which matches the branch, is used. The other difference from existing hybrid predictors lies in the handling of table updates. Table 7 shows the update rules for the Target Cache and Cascaded predictors. For the Target Cache, when an indirect branch is resolved, the Target cache (second stage predictor) is updated with its target address. Otherwise, updates are done in the first stage predictor (BTB) only.

Predictors		Update Rules
Target Cache		- When an indirect branch is resolved, the 2nd stage is updated with its target address. - For a first-miss, update of the 2nd stage is not allowed.
Cascaded predictor	Strict Filter	- When an indirect branch is resolved, the 1st and 2nd stages are updated. - For a first-miss, update of the 2nd stage is not allowed.
	Leaky Filter	- When an indirect branch is resolved, the 1st and 2nd stages are updated. - For a first-miss, update of both the 1st and 2nd stages is allowed.

Table 7. Update rules for the Target Cache and Cascaded predictors.

Table 7 shows the update rules for the Target Cache and also the strict and the leaky filters which are two variations of the Cascaded predictor. The Cascaded predictor can reduce the table size effectively by using a small-sized BTB as a filter. Since the first stage works as a filter to separate indirect branches from amongst all branches, the

second stage is used to store indirect branches that have multi-targets. Therefore, the accuracy of a 2-stage predictor is much higher than that of a single-scheme. The difference between the strict and leaky filters is that the leaky filter allows new second stage entries on a first-miss while the strict filter does not [Driesen & Hölzle '98B].

The differences between indirect and conventional hybrid branch predictors are:

- Indirect branch predictors record branch targets instead of directions in the table;
- Indirect branch predictors employ different selection mechanisms other than a 2-bit saturating counter;
- Indirect branch predictors have different table update rules instead of just updating both predictors simultaneously.

3.3 GoStay2 Branch Predictor

Section 3.2 described several indirect branch predictors in detail. Both the Target cache and Cascaded predictor can reduce the indirect misprediction rate considerably over a BTB-based predictor. Among them, the leaky filter of the Cascaded predictor offers the most effective misprediction rate performance for indirect branches [Kalamatianos & Kaeli '98][Driesen & Hölzle '98B]. However, the leaky filter has some problems that degrade system performance:

- Conflict misses – If a prediction table such as BTB has small entries (say, less than 512 entries), conflict misses might increase the misprediction rate considerably;
- Inefficient predict rules – If a branch address is found at both stages, the second stage has priority for prediction. If the first stage has a correct target address and the second

stage has an incorrect target address, then the assumed priority of the second stage always causes a misprediction.

- Inefficient update rules – If a predicted target address is wrong, then the resolved target address of the branch address is updated in both stages. This also causes a misprediction if the replaced target address is needed for a following branch.

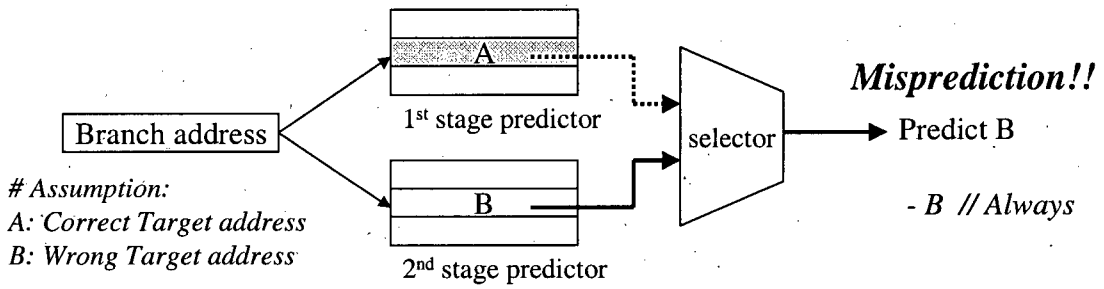
In order to resolve the above problems, this section presents a new hybrid branch predictor.

3.3.1 An overview of a GoStay2 predictor

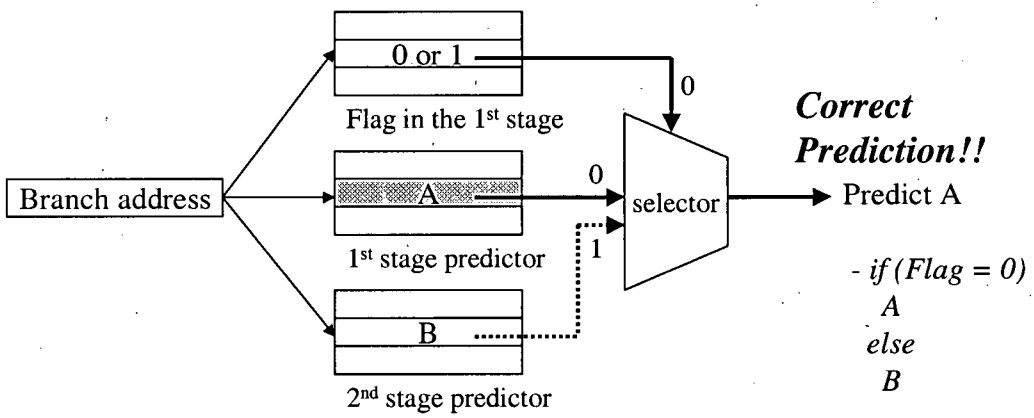
As we discussed in section 3.2, the basic operation of 2-stage hybrid predictors can be divided into the three parts comprising indexing, predicting, and updating. For predicting and updating, each 2-stage hybrid predictor has its own predict and update rule to predict a target address and update a resolved target address.

Figure 37(a) shows that, in a conventional 2-stage branch predictor, if the first stage has a correct target address (A) but the second stage has a wrong one (B), then the prediction (B) leads to misprediction since the second stage always takes priority of prediction.

Figure 37(b) shows the basic operation of a GoStay2 predictor, which can reduce mispredictions effectively. In a GoStay2 predictor, the prediction will be made according to the flag in the first stage. In Figure 37(b), since the flag is '0', the prediction (A) is made with the target address in the first stage (A), which leads to correct prediction. The flag is updated to '0' or '1' according to the update rule (refer to section 3.3.3).



(a) conventional 2-stage branch predictors



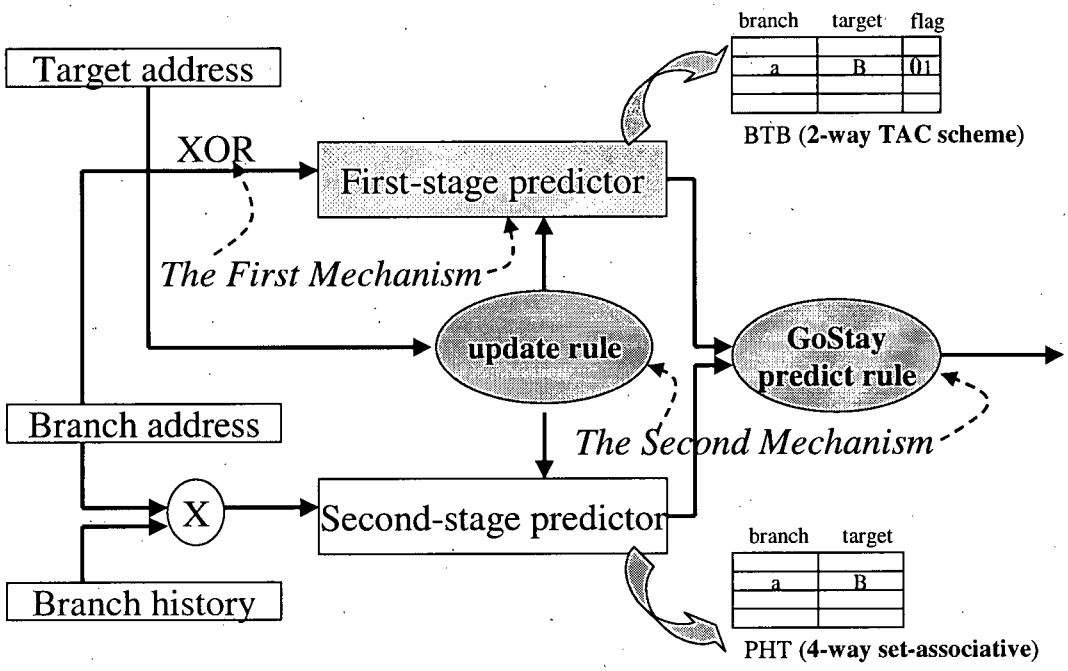
(b) GoStay2 branch predictors

Figure 37. The basic operations of conventional 2-stage and GoStay2 branch predictors.

Figure 38 shows the overview of the proposed branch predictor called the GoStay2, which has a different operation from other 2-stage hybrid branch predictors such as the Target Cache and Cascaded predictor. *'GoStay2' implies GoStay predict and update rules, as well as a 2-bit flag in the first stage.*

In the GoStay2, the indexing function for the first stage is different from the other predictors (refer to section 3.3.2), but that of the second stage is the same as the others shown in Figure 37. For predicting, the GoStay2 predictor provides a new selection scheme called the GoStay predict rule (refer to section 3.3.3). Our experiment shows that the GoStay predict rule is more accurate than the leaky filter. Finally, for updating, this section introduces a new replacement policy for the first stage and a new update rule for

both stages by using a 2-bit flag in the first stage. The first bit of the flag is for the Bank-originated Pseudo-LRU (BoPLRU) replacement policy [Chu & Ito '00], and the second bit is for the GoStay. Figure 38 shows all the differences mentioned above as two mechanisms.



** BTB: Branch Target Buffer, TAC: Thrashing Avoidance Cache, PHT: Pattern History Table

Figure 38. The overview of the GoStay2 branch predictor.

For the first mechanism in Figure 38, the GoStay2 Predictor uses a new cache scheme developed by Chu and Ito ('00) instead of a 4-way set-associative for the first stage to reduce conflict misses. This new scheme called the 2-way TAC employs 2-way banks and the XOR mapping function (XOR). The XOR is used for indexing the 2-way TAC scheme by using a branch address. Bodin and Seznec ('95) defined the XOR for 2-way banks such that each bank consists of 2^n cache lines of 2^c bytes, where σ is the perfect-

shuffle on n bits, so that the data block at memory address $A_3 2^{c+2n} + A_2 2^{n+c} + A_1 2^c$ may be mapped:

- on a cache line $A_1 \oplus A_2$ in cache bank 0
- or on a cache line $\sigma(A_1) \oplus A_2$ in cache bank 1

The 2-way TAC contains a branch and target address along with a 2-bit flag per cache line, which is added one more bit from the 2-way TAC of Chu and Ito ('00). The main function of this scheme is to place a group of branch instructions into a bank according to the BSL (Bank Selection Logic) and the BoPLRU replacement policy and is explained in more detail in section 3.3.2. The combination of BSL and BoPLRU guarantees that recent groups of branches can be retained in each bank safely.

For the second mechanism in Figure 38, to improve the inefficient predict and update rules, the second bit of the 2-bit flag is used to implement the GoStay predict and update rule if both stages have a branch address: If the second bit is '1', a target address of the second stage is used (Go). Otherwise, a target address of the first stage is used (Stay). The GoStay2 predictor works the same as the leaky filter if the bit is '1'. This bit is set to '0' whenever a branch address is found in the first stage only and the predicted target address is correct. In other words, if the second bit of the 2-bit flag is '0', then the branch address is indirect, and the target address was correct for the previous prediction.

3.3.2 The 2-way TAC scheme for the BTB – The first mechanism

As we discussed before, the first stage employs the 2-way TAC scheme to reduce conflict misses for small-sized (say, less than 512 entries) tables. The first mechanism is defined as two functions for this scheme (Figure 39), namely indexing (XOR mapping) and updating.

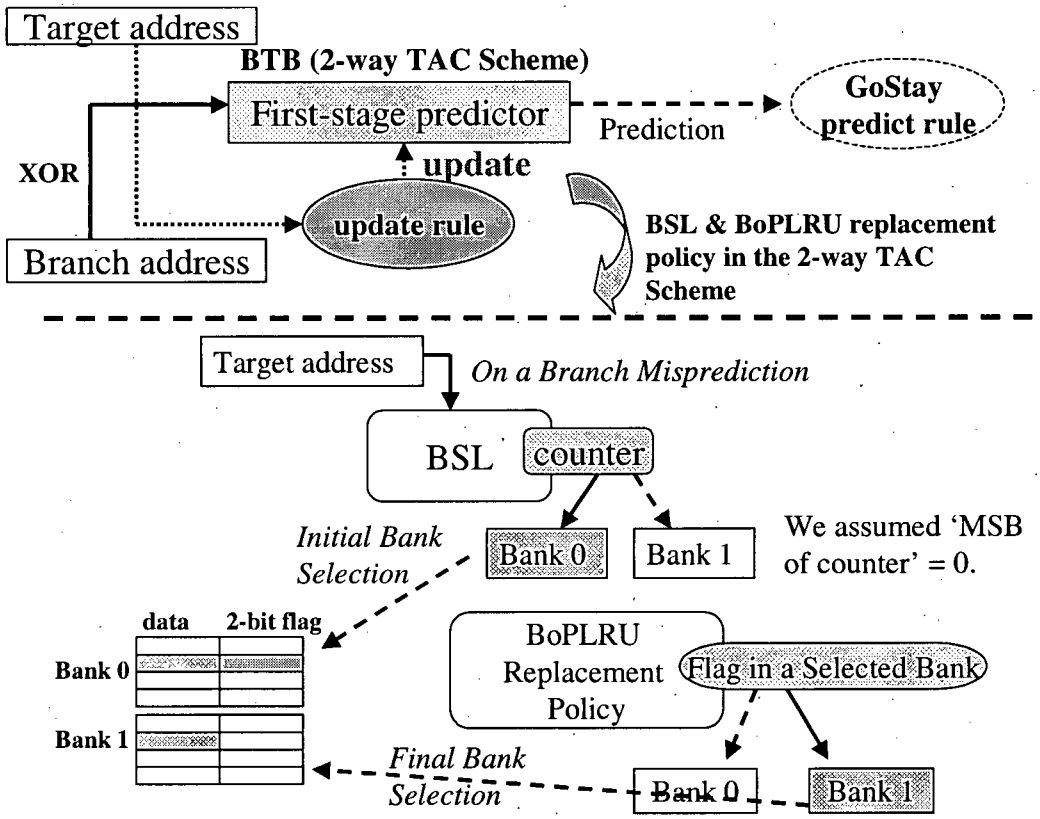


Figure 39. The operation of the first mechanism. data = branch address + target address.

For the indexing function, the two banks of the first stage predictor are accessed simultaneously with two different XOR mapping functions as we discussed in the previous section. Since Gonzales et al. ('97) and Seznec ('93) have shown that the XOR

works well for reducing conflict misses, the GoStay2 employs this mapping function for this scheme.

For updating function, if the GoStay update rule (refer to section 3.3.3) selects the first-stage predictor to update resolved branch/target addresses, they are written into the selected bank of the first-stage according to the value of the first bit of the 2-bit flag. In Figure 39, the BSL selects a bank initially on a miss according to a fixed frequency of the procedure call instructions. The BSL employs a n-bit counter for counting the occurrences of call instruction. For example, if n = 2, then the first bit (Most Significant Bit) of the counter toggles every second procedure calls and the toggled first bit shows a selected bank. Therefore, if an instruction in the first group is initially placed in bank0, then an instruction in the third group is placed into bank1. In this chapter, it is assumed that the first stage employs a 2-bit counter.

After the BSL selection in Figure 39, the BoPLRU determines the final bank for updating a line as a correction mechanism by checking the first bit of the flag for the selected cache line [Chu & Ito '00]. When the first bit of the 2-bit flag is '1', the branch/target addresses are written into bank0, and the first bit is changed to '0'. Otherwise, the branch/target addresses are written into bank1, and the first bit is changed to '1'. In the Figure 39 example, it is assumed that bank0 of the first stage is selected for updating by the BSL. Then, the BoPLRU policy works according to the status ('1' or '0') of the first bit in the 2-bit flag. In Figure 38, bank 1 is selected for updating a cache line since the first bit of the 2-bit flag is '0', then the 2-bit flag is changed from '**0**1' to '**1**1' [Chu & Ito '00].

This mechanism helps improve indirect misprediction rates by reducing conflict misses in a small-sized, less than 512 entries, first stage predictor table such as BTB.

3.3.3 The GoStay predict and update rule – The second mechanism

For the second stage, like other 2-stage hybrid branch predictors, the GoStay2 can use a pattern- or path-based history xored with low-order bits of a branch address as an indexing function. There are two functions in the second mechanism: the GoStay predict rule and update rule.

3.3.3.1 GoStay predict rule

Figure 40 shows that each stage is examined as to whether the current branch address is in the table or not. There are three possible cases:

1. If there is no matched branch address in either stage, then this is a case of 'not taken'. Therefore, no prediction occurred;
2. If there is one matched branch address between two stages, then the prediction occurs with the target address of a matched stage;
3. If both stages have the same matched branch address, the prediction will be determined according to the GoStay predict rule in Figure 40.

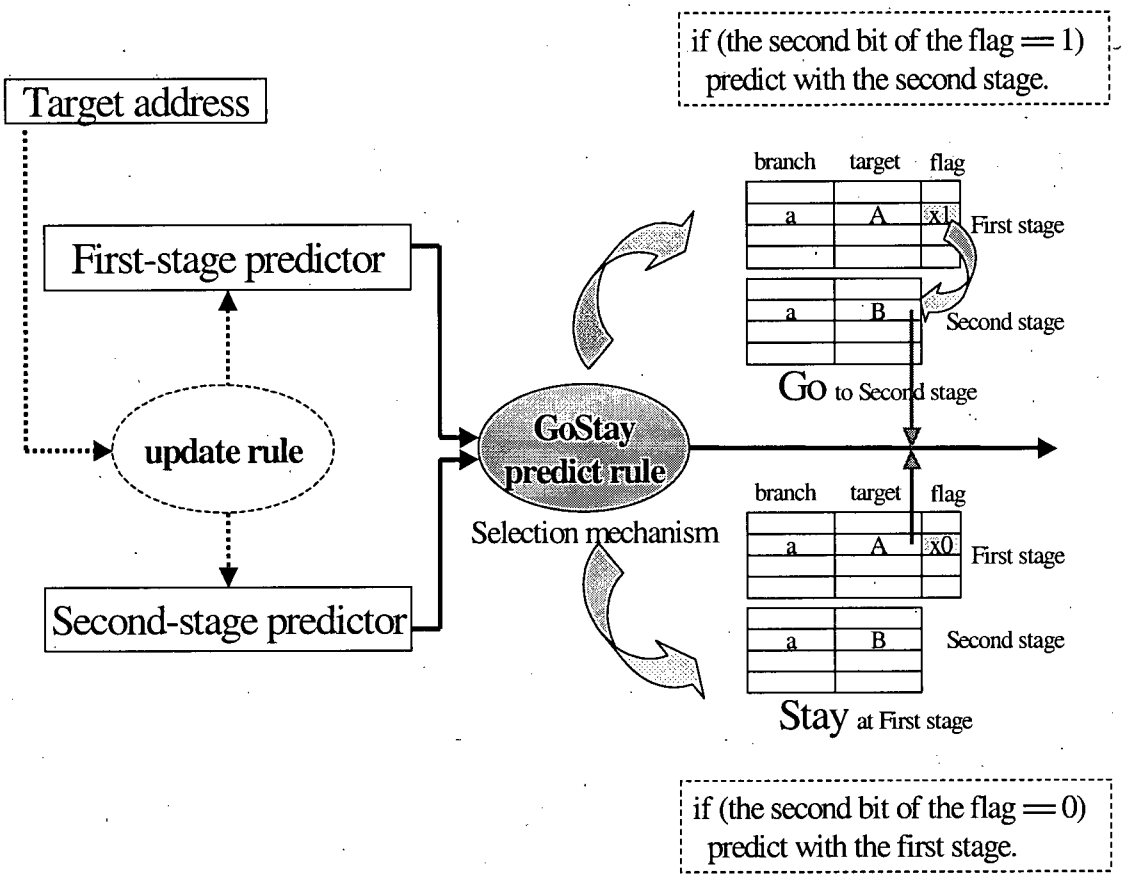


Figure 40. The GoStay predict rule of the second mechanism.

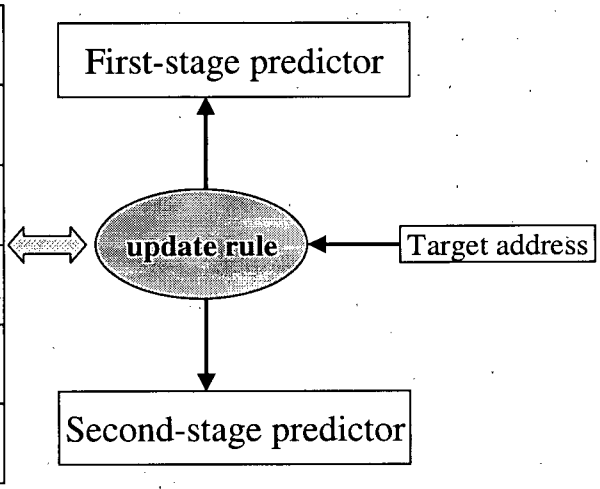
The goal of the GoStay predict rule is to reduce mispredictions caused by wrong target addresses of the second stage. This rule works only when both stages have the same branch address. The detailed operations in the 'GoStay predict rule' of Figure 40 are:

1. If the second bit of the 2-bit flag in the first stage is '1', then the prediction will be done with the target address of the second stage (Go);
2. If the second bit of the 2-bit flag in the first stage is '0', then the prediction will be done with the target address of the first stage (Stay).

3.3.3.2 Update rule

Update rule for the GoStay2

Prediction with	Results	Update	Flag (2nd bit)
None	None	Both stages	1
First stage	Correct	None	0
First stage	Incorrect	Both stages	1
Second stage	Correct	None	1
Second stage	Incorrect	Second stage	1



Prediction with: Predicting with a target address of a selected bank.
Results: Prediction results after the execution.
'Correct' means correct predictions and 'incorrect' means misprediction.
Both stages means both the first and second stages.
Flag is the 2nd bit of the 2-bit flag in the first-stage predictor.

Figure 41. Update rule of the second mechanism.

Figure 41 shows the update rule after the branch instruction is resolved. The branch history register will be updated according to the branch resolution. There are three cases for updating both stage predictors.

First, *in case of no prediction*, both stages are updated with a new branch and target address. This is the case of a first-miss. Both the leaky filter and the GoStay2 permit updating of the second stage table for a first-miss. In addition to this, the GoStay2 sets the second bit of the 2-bit flag in the first stage as '1'. This second bit of the 2-bit flag is always set to '1' whenever the second stage is updated. Therefore, if the second bit is '1', the GoStay2 works like the leaky filter in this case.

Secondly, *in the case of prediction with a target address in the first stage*, the update can also be varied according to the branch resolution:

1. If it is a correct prediction, the second-bit of the 2-bit flag is set to '0';
2. If it is an incorrect prediction, both stages are updated. The second bit of the 2-bit flag is set to '1'.

Lastly, *in case of prediction with a target address in the second stage*, the update will be varied according to the branch resolution:

1. If it is a correct prediction, no update is required;
2. If it is an incorrect prediction, the target address of the second stage is updated since the branch address is indirect.

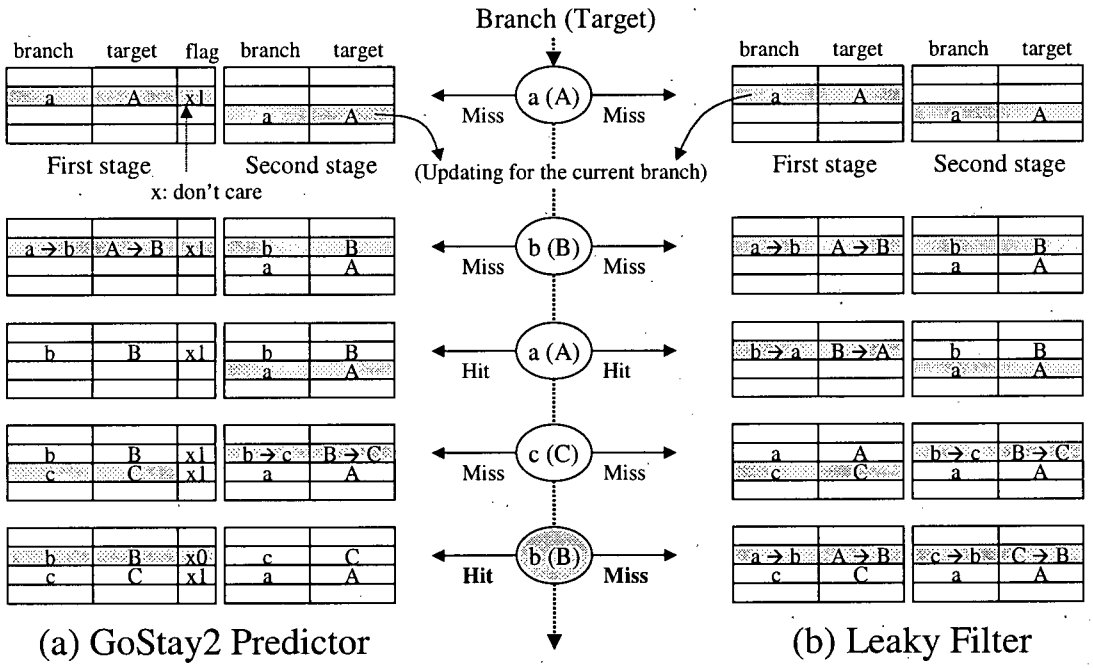
3.3.4 Benefits of the GoStay2 branch predictor

Figure 42 shows an example of committed target addresses, which compares the update processing between the GoStay2 and leaky filter. The assumptions for the branch addresses are:

- a(A), b(B), and c(C) show the 'branch address (target address)' in each table;
- In the first stage, a(A) and c(C) have the same branch indexing but different target address;
- In the second stage, b(B) and c(C) have the same branch indexing but different target address.

Figure 42(a) shows how the GoStay2 works with the flow of example target addresses:

- a(A) is placed into the first stage with the flag set to 'x1' and into the second stage since a(A) is assumed to be a first-miss. The 'x' of 'x1' means 'don't care' since the first bit is set by the replacement policy of the 2-way TAC scheme for the first stage. The second bit of the flag is set to 1 since the second stage is updated. (Miss);
- b(B) conflicts with a(A) in the first stage because of the same branch indexing, so the target address is changed A to B in the first stage. b(B) is placed into the second stage without conflict. The second bit of the flag for b(B) in the first stage is set to 1 since the second stage is updated. (Miss);
- a(A) is found in the second stage. Therefore, there is no update for a(A). (Hit);
- c(C) is placed into the first stage with the flag set to 'x1' since c(C) is assumed to be a first-miss. b(B) in the second stage is changed to c(C) since there is a conflict between b(B) and c(C) in the second stage according to the assumption. The second bit of the flag is set to 1 since the second stage is updated. (Miss);
- b(B) is found in the first stage only, so that the second bit of the flag is set to 0 without further update according to the update rule. Therefore, the GoStay2 can save b(B) in the first stage longer. *In the case of the next occurrence of b(B), the GoStay2 can predict the target address correctly.* (Hit).



Assumptions:

1. Same branch indexing in the first stage: {a(A) and c(C)}.
2. Same branch indexing in the second stage: {b(B) and c(C)}.
3. a(A), b(B), and c(C) → branch address (target address).

Figure 42. A comparison of the update processing between the GoStay2 and the leaky filter.

Figure 42(b) explains how the leaky filter works with the flow of example target addresses:

- The update process of the first a(A) and b(B) are just like the GoStay2 except for the updating the flag bit since there is no flag bit in the leaky filter. (Two Misses: a(A) and b(B));
- a(A) is found in the second stage only. Therefore, b(B) in the first stage is changed to a(A) since there is a conflict between a(A) and b(B) in the first stage according to the assumption. (Hit);

- c(C) is placed into the first stage. b(B) in the second stage is changed to c(C) since there is a conflict between b(B) and c(C) in the second stage according to the assumption. (Miss);
- b(B) is placed into the first and second stage. a(A) in the first stage is changed to b(B) since there is a conflict between a(B) and b(B) in the first stage. c(C) in the first stage is changed to b(B) since there is a conflict between c(C) and b(B) in the second stage. (Miss).

From Figure 42, the benefits of the GoStay2 predictor can be determined. The main feature of indirect branches is more than one target address for an indirect branch. Since the target of an indirect branch can vary with every dynamic instance, the use of a history pattern is needed to select the correct one among several target addresses stored in the prediction tables. If an indirect branch has several targets, then it can be assumed that each target has its own history. Therefore, if each target is preserved for longer than before, then the indirect mispredictions can be reduced effectively. The GoStay2 predictor can do this by using a 2-bit flag effectively as shown in Figure 42. *The benefits of the GoStay2 predictor result from retained target addresses that have a different history pattern since they can stay longer than can other 2-stage hybrid predictors in the prediction tables.*

3.4 Experimental environment

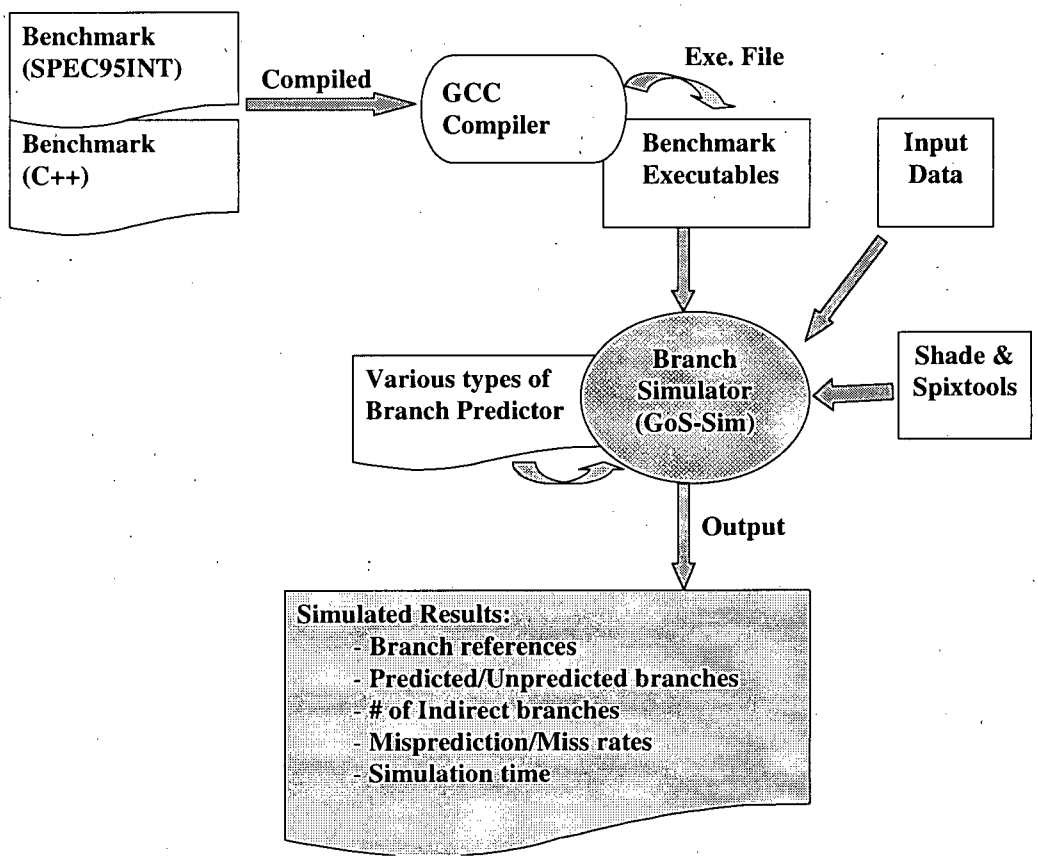


Figure 43. Experimental methodology

Figure 43 shows an overview of our simulation methodology:

- First, SPECint95 and C++ programs were compiled by using a compiler (GNU gcc 2.6.3 and 2.7.2 are used); and

- Second, the GoS-Sim (branch prediction simulator) ran each executable benchmark with its input data. GoS-Sim was developed by using the Shade and SpixTools. Shade and SpixTools are tracing and profiling tools developed by Sun Microsystems. Shade executes all the program instructions and passes them onto the branch

prediction simulator, GoS-Sim. SpixTools are used for collecting the information for static instructions. GoS-Sim not only simulates most indirect branch predictors such as the BTB-based Target Cache and Cascaded Predictor, but it also runs several XOR mapping functions and replacement policies such as the LRU (Least Recently Used) and the Pseudo LRU, etc. The simulator for the proposed predictor was added into the GoS-Sim. Finally, Outputs such as misprediction rates, the number of control transfer and procedural call/return instructions, etc. were collected.

In Figure 43, *Shade* is a tool that dynamically executes and traces SPARC v9 executables [Cmelik & Keppel '93]. One can specify the exact trace information that is desired by using Shade. That means the trace information can be dynamically handled in any manner. Detailed information for every instruction and opcode can be collected dynamically. For example, the data for the total number of call instructions, program counter, opcode fields, etc. can be obtained. This information is used for our simulation tool, GoS-Sim.

3.4.1 Benchmarks

Table 8 describes the benchmark programs in detail. Five of the SPECint95 programs were used for our simulation – go, xlisp, m88ksim, gcc, and perl. These are the same programs used in [Driesen & Hölzle '98B][Radhakrishnan & John '98]. In order to compare our results with them, the SPECint95 instead of the SPEC2000 were used for our simulation. The next suite of programs is written in C++ and has been used for

investigating the behavior between C and C++ [Calder et al. '94A][Hölzle & Ungar '94]. These programs are ixx, eqn, and deltablue.

Program	Input	Description
SPEC95 CINT: C Programs		
go	2stone9.in	Plays the game Go against itself
xlisp	train.lsp	Lisp interpreter
m88ksim	ctl.raw	Simulates the Motorola 88100 processor
gcc	amptjp.i	Compiles pre-processed source
perl	Scrabbl.pl scrabble.in	Performs text and numeric manipulations
Suite of C++ Programs		
ixx	object.h som_plus_fresco.idl	IDL parser generating C++ stubs
eqn	eqn.input.all	Type-setting program for mathematical equations
deltablue	3000	Incremental dataflow constraint solver

Table 8. Benchmark descriptions

Table 9 provides a description of the run-time characteristics of the benchmarks. Dynamic instructions represent the number of instructions executed by each program. It also shows the number of control transfer instructions. Control transfer instructions are divided into three groups such as branches, direct calls, and indirect jumps. It shows that the rate for control transfer instructions (except branches) in object-oriented languages (C++) is two to three times higher than for traditional languages (C).

Program	Dynamic instructions	# of control transfer instructions			
		Total	Branches	Calls	Jumps /indirect calls /returns /trap returns
go	584,163K	82,253K	69,163K	1,611K	11,479K
xlisp	189,185K	43,643K	30,288K	7,971K	5,384K
m88ksim	851K	196K	171K	17K	9K
gcc	250,495K	53,190K	43,711K	5,204K	4,274K
perl	630,281K	130,746K	88,162K	26,110K	16,473K
ixx	31,830K	7,258K	4,731K	1,405K	1,121K
eqn	58,401K	12,080K	9,033K	1,999K	1,048K
deltablue	42,149K	9,997K	5,122K	1,478K	3,397K
C Mean (harmonic)	4,210K	970K (23.04 %)	842K (86.86 %)	82K (8.42 %)	42K (4.38 %)
C++ Mean (harmonic)	41,514K	9,358K (22.54 %)	5,800K (61.98 %)	1,589K (16.98 %)	1,401K (14.98%)

Table 9. Benchmark characteristics

3.5 Experimental results

3.5.1 Implemented branch Predictors

We simulated several indirect branch predictors, compared their misprediction rates, and analyzed misprediction rates. For 2-stage hybrid predictors, most mispredictions occur when both stages have a current address but not a correct target address. Therefore, the analysis of indirect branch mispredictions will be done by examining the misprediction rates according to the cases whether both stages have a correct target for a branch or not. The total number of entries for the PHT (the second stage predictor) is kept

constant (set to 512 and 1K entries) while BTB (the first stage predictor) varies from 8 entries to 4K entries. The PHT is used for the prediction table for the second stage, which stores target addresses instead of directions (taken/not taken).

The implemented predictors are:

- **BTB** (Branch Target Buffer): 4-way set-associative, indexing with the low-order bits of a branch address. The table sizes vary from 8 entries to 4K entries.
- **TC** (Target Cache): The first stage is a BTB and the second stage is a 512- or 1K-entry of the Target Cache (similar to the PHT, 4-way set-associative) using a gshare scheme. The 9-bit (512-entry) or 10-bit (1K-entry) history registers record pattern-based history information. The BTB and TC are examined simultaneously for a branch. If the BTB detects an indirect branch, the selected TC entry is used for target prediction. After resolving the indirect branch, the TC only can be updated with its target address.
- **SF** (Strict Filter for Cascaded predictor): The hardware mechanism is similar to the TC. The main difference to the TC is the handling of the table updates. SF only allows branches into the second-stage predictor if the first predictor mispredicts.
- **LF** (Leaky Filter for Cascaded predictor): The hardware mechanism is similar to the SF. The difference to the SF is that the LF allows new second-stage entries on first-misses in addition to the update rule of the SF.
- **GoS** (GoStay2): The first stage is a 2-way TAC scheme. It contains branch and target addresses with a 2-bit flag. The prediction table is accessed with the XOR mapping functions. The second stage is similar to the LF. The other differences

from TC, SF, and LF are the new predict and update rules which depend on the flag of the first stage predictor (refer to section 3.3).

The main goal of this section is to compare the performance of the selection mechanism and update rule of the GoStay2 with other 2-stage hybrid predictors such as TC, SF, and LF. Therefore, the above predictors were implemented with the same indexing function for the second stage predictor.

For the pattern history of the second stage, the history register records the direction of the previous conditional branches. Nair ('95) showed that a path-based predictor with two-bit partial addresses attained prediction rates similar to a pattern-based predictor with taken/not taken bit (similar hardware budgets) [Driesen & Hölzle '98A]. Therefore, the gshare scheme was implemented by using the pattern-based history only because benchmark programs were traced with both direct and indirect branches.

Our simulation results generated misprediction rates for several of the predictors that are a little bit higher than previously reported: Target Cache [Chang et al. '97] and Cascaded predictor [Driesen & Hölzle '98B]. This could have happened because all kinds of control transfer instructions were traced to examine the predictability of various cases such as branches, procedure calls/returns, indirect jumps, etc. However, the Target Cache in Chang et al. ('97) recorded the target for each indirect jump encountered only, while the Cascaded predictor in Driesen and Hölzle ('98B) excluded procedure returns with the assumption that they could be predicted accurately with a return address stack.

3.5.2 Indirect Branch Instructions

According to Chang et al. ('97), control transfer instructions can be categorized into three groups: direct conditional, direct unconditional or indirect unconditional branch instructions. In case of direct conditional/unconditional branches, they have a single-target which can be predicted with reported hit-ratios of up to 97% [Yeh & Patt '93]. However, indirect branch instructions commonly have multi-target addresses. The multi-target addresses of an indirect branch are created dynamically while the program is executed.

Program	Type	Dynamic instructions	Control Flow Instructions					
			Total		Conditional branches		Indirect branches	
			num.	%	num.	%	num.	%
xlisp	C	189,185K	43,643K	100	30,288K	69.40	4,076K	9.34
ixx	C++	31,830K	7,258K	100	4,731K	65.19	538K	7.42
perl	C	630,281K	130,746K	100	88,162K	67.43	7,656K	5.97
gcc	C	250,495K	53,190K	100	43,711K	82.18	3,177K	5.97
eqn	C++	58,401K	12,080K	100	9,033K	74.78	547K	4.53
m88ksim	C	851K	196K	100	171K	87.02	4K	2.27
go	C	584,163K	82,253K	100	69,163K	84.09	548K	0.67
deltablue	C++	42,149K	9,997K	100	5,122K	51.24	554K	5.54

Table 10. Comparisons for the percentages of conditional and indirect branches.

Table 10 shows the percentages of conditional branches and indirect branches for the benchmark programs. In the case of conditional branches, the percentages of the C programs are higher than the C++ programs in Table 10. In addition, in the case of

indirect branches, 'xlisp' shows the highest (9.34%) and 'go' has the lowest (0.67%) among all benchmark programs.

Program	Type	Lines of code	Inst./ ind.	Cond./ ind.	Descriptions
xlisp	C	4,700	46.42	7.43	Using for all averages in section 3.5.4 and 3.5.5.
ixx	C++	11,600	59.12	8.79	
perl	C	21,400	82.33	11.52	
gcc	C	130,800	78.85	13.76	
eqn	C++	8,300	106.74	16.51	
m88ksim	C	12,200	190.54	38.25	
go	C	29,200	1,065.49	126.15	Low indirect branches
deltablue	C++	500	76.07	9.24	Small-sized program

Table 11. The relevance of indirect branches by comparing lines of code, inst./ind. (instructions/indirect branch), and cond./ind. (conditional branches/indirect branch).

Table 11 shows that the relevance of indirect branch is related to the lines of code, by the number of the instructions per indirect branch, and by the number of conditional branches per indirect branch. Three groups emerge: first, four of the SPECint95 benchmarks and two C++ benchmarks execute fewer than 200 instructions per indirect branch; second, one of the SPECint95 benchmarks execute more than 1,000 instructions per indirect branch; third, one C++ benchmark has less than 500 lines of code, a small-sized benchmark program.

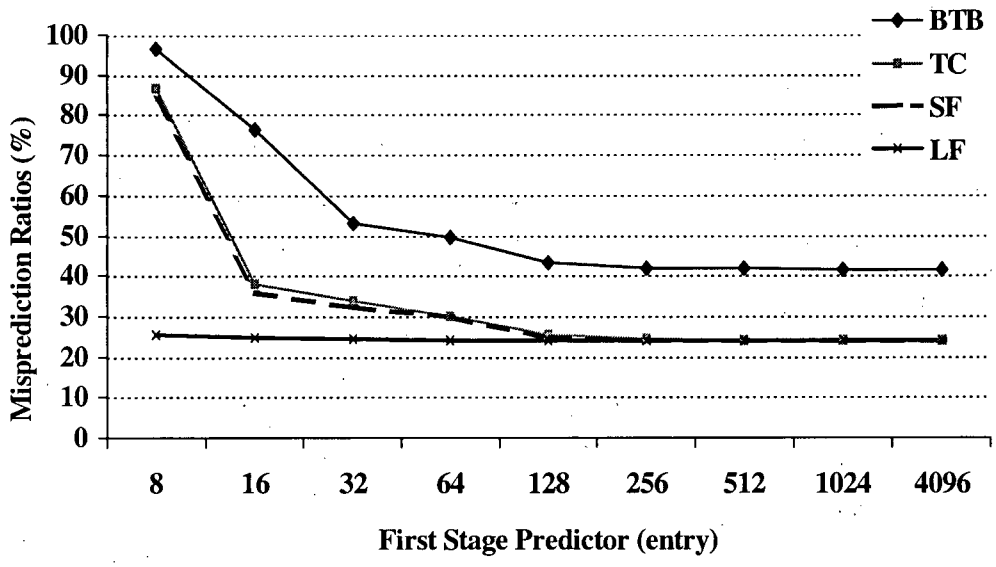
In Table 11, the first group of benchmark programs, from xlisp to m88ksim, executes from 7 to 38 conditional branches per indirect branches. They are a good choice for evaluating indirect branch predictors since the frequency of indirect branches is spread

from higher (7.43) to lower (38.25). The results of the misprediction rates for the first group will be used for all averages in section 3.5.4 and 3.5.5.

For "go", since the impact of indirect branch prediction is very low, it will be excluded from all averages in sections 3.5.4 and 3.5.5.

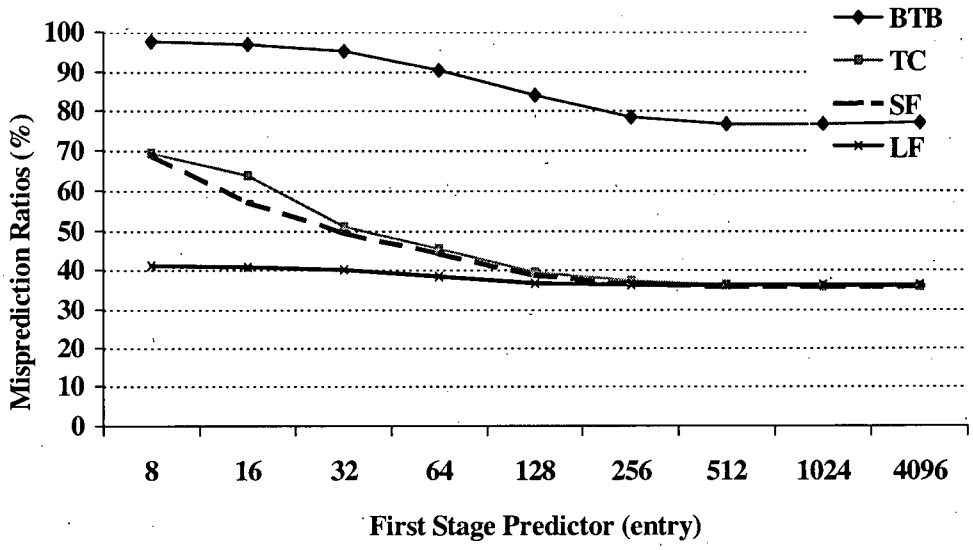
For "deltablue", even though it executes fewer than 10 conditional branches per indirect branch, most indirect branch mispredictions will be reduced by using an appropriate size of the LF and GoS because of the small-sized (less than 500 lines) program. Therefore, it also is excluded from all averages like the second group.

3.5.3 Conventional indirect branch predictors

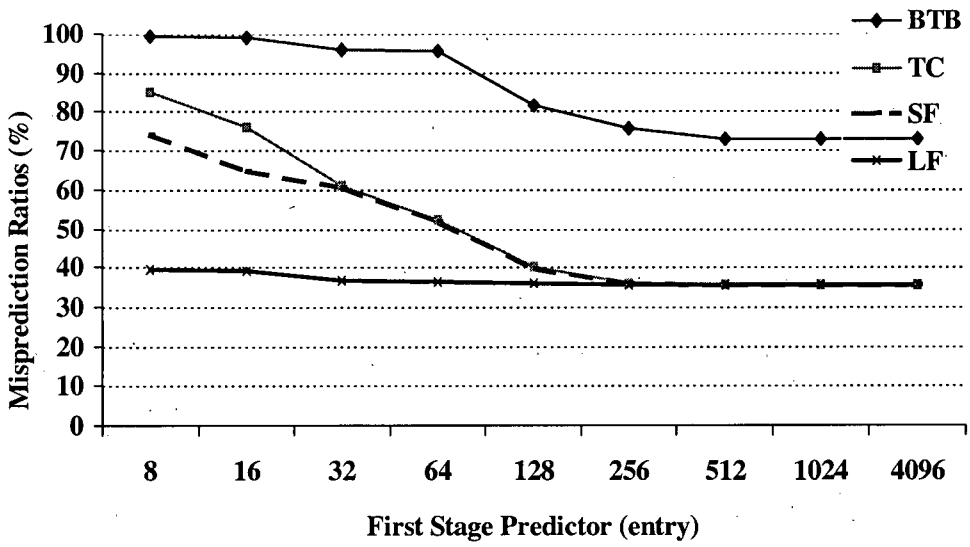


(a) xlisp (C program)

Figure 44. The comparison of misprediction rates according to BTB sizes for indirect branch predictors. The second stage is a table with 512 entries (4-way).

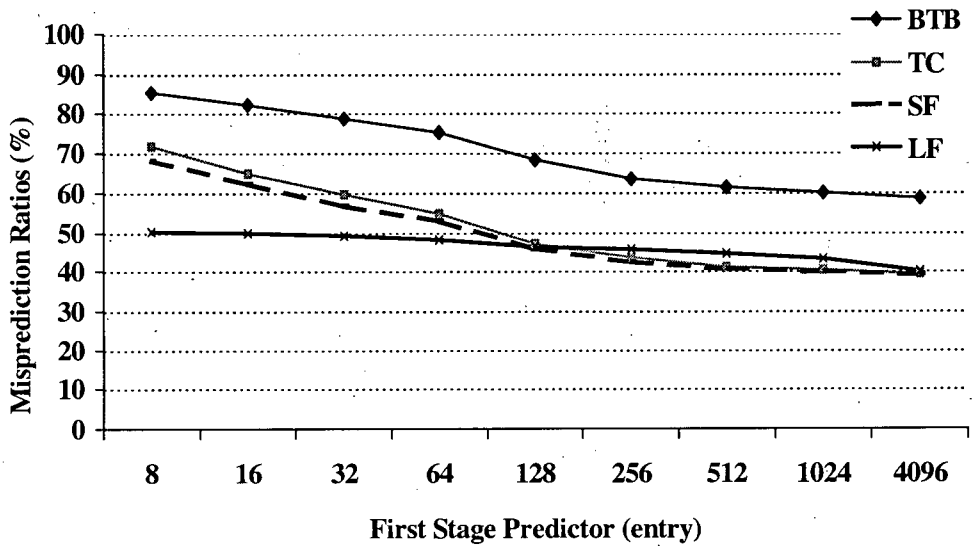


(b) ixx (C++ program)

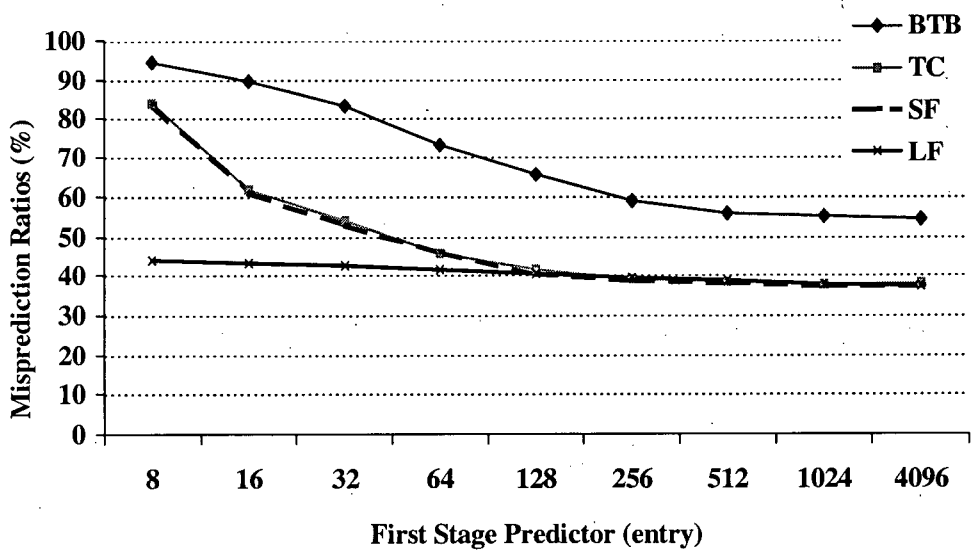


(c) perl (C program)

Figure 44. (continued) The comparison of misprediction rates according to BTB sizes for indirect branch predictors. The second stage is a table with 512 entries (4-way).

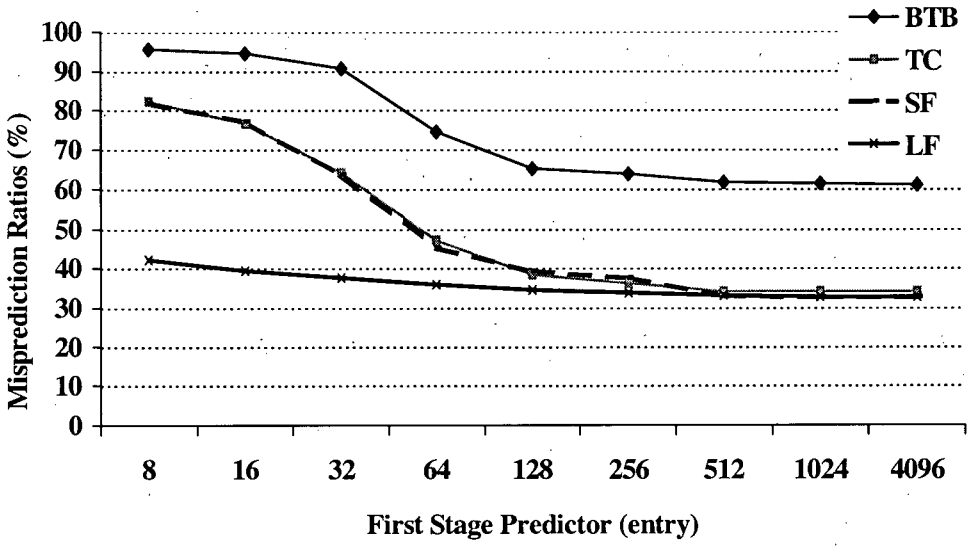


(d) gcc (C program)

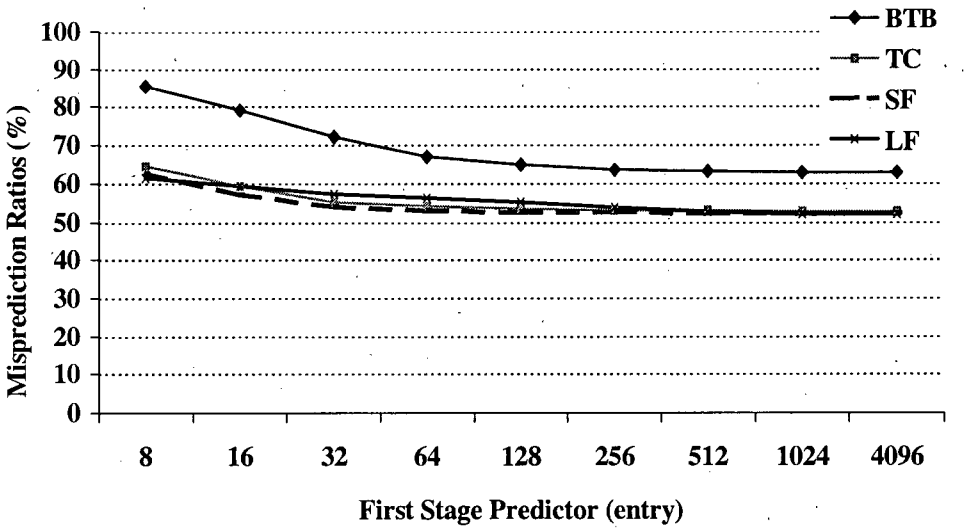


(e) eqn (C++ program)

Figure 44. (continued) The comparison of misprediction rates according to BTB sizes for indirect branch predictors. The second stage is a table with 512 entries (4-way).

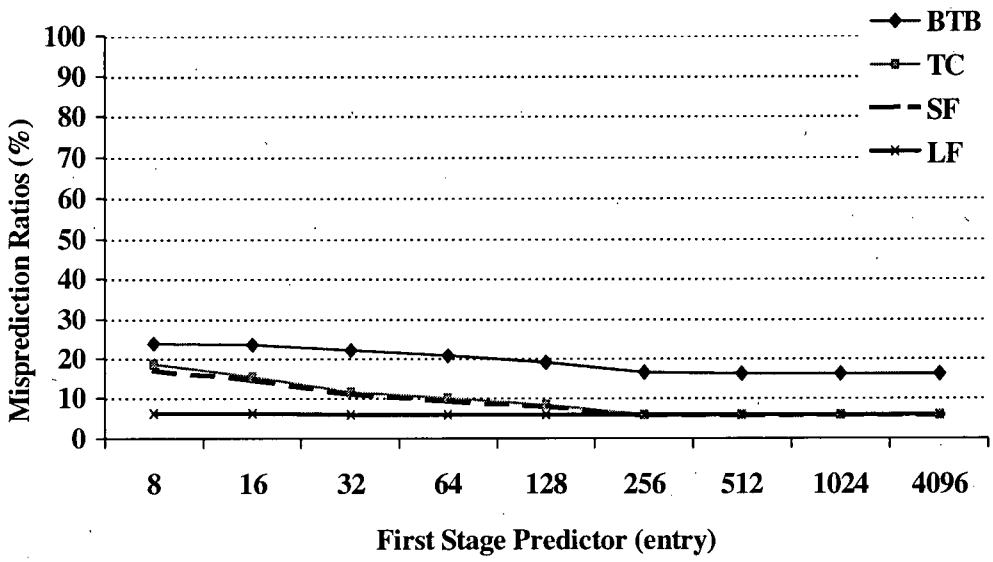


(f) m88ksim (C program)



(g) go (C program)

Figure 44. (continued) The comparison of misprediction rates according to BTB sizes for indirect branch predictors. The second stage is a table with 512 entries (4-way).



(h) deltablue (C++ program)

Figure 44. (continued) The comparison of misprediction rates according to BTB sizes for indirect branch predictors. The second stage is a table with 512 entries (4-way).

This section determines the most effective branch predictor among the BTB, TC, SF, and LF. Eight benchmark programs are examined according to the size of the tables of the first stage. For hybrid predictors (the TC, SF, and LF), the second stage has a table with 512-entries and is 4-way set-associative.

Figure 44 shows the misprediction rates for four predictors using eight benchmark programs:

1. Figure 44(a), (b), (c), (e), and (f) showed that the LF is the most effective among other previous predictors. If the size of the BTB is less than 128 entries, the LF reduces misprediction rates much better than other previous predictors do. Otherwise,

the LF works only slightly better than most others, and does much better than the BTB;

2. Figure 44(d) showed that the LF is the most effective if the size of the BTB is less than 128 entries, otherwise the SF works slightly better than the LF. This result can occur because *gcc* contains a large number of static branches in its working set. This large set can cause interference in the second stage predictor, reducing the ability to make accurate predictions [Chang et al. '95];
3. Figure 44(g) showed that the SF is the most effective if the BTB is less than 512 entries, otherwise the LF works slightly better than the SF. This result can occur because *go* contains very low indirect branches compared to other benchmark programs.
4. Figure 44(h) showed that the LF is the most effective among other predictors. However, the overall misprediction rates are much lower than those of other benchmark programs because *deltablue* has only 500 lines of code (a small-sized program).
5. The SF works slightly better than the TC for all sizes of BTB-entries;
6. In general, hybrid predictors (the TC, SF, and LF) are much more effective than a single-scheme predictor such as the BTB for all sizes of BTB-entries.

From the above results, the LF is determined as the most effective indirect predictor. Driesen and Hölzle ('98B) also showed that the misprediction rate for the LF is much better than the SF for small-sized filters (BTB) such as 128 entries or less; otherwise, the LF is slightly better than the SF. Kalamatianos and Kaeli ('98) also showed that the LF,

with a filter (BTB) of 128 entries, suffered fewer mispredictions than the TC. In the following sections, the LF will be used as the representative indirect predictor for comparing the misprediction rates with the GoS.

3.5.4 Misprediction rates for indirect branches between the LF and GoS

In this section, the indirect misprediction rates are compared between the LF and GoS. As we discussed before, there are some differences between the LF and GoS:

- For the cache scheme of the first stage, the LF uses a 4-way set-associative, but the GoS employs a 2-way TAC scheme. However, for the second stage, both predictors use the PHT (512- or 1K-entry, 4-way) with the same indexing function (the *gshare scheme* for this thesis).
- For the selection mechanism, if both stages have target addresses, the second stage of the LF takes precedence for prediction as in the TC and SF. However, for the GoS, even if both stages have target addresses, the prediction will be taken according to the status of the second bit of the flag in the first stage. For example, if the bit is 0, the first stage takes precedence, otherwise the second stage does.
- For the update rule, the LF allows an entry to the second stage for a first-miss or indirect branches. However, the GoS updates the two stages according to the update rule according to the status of the flag.

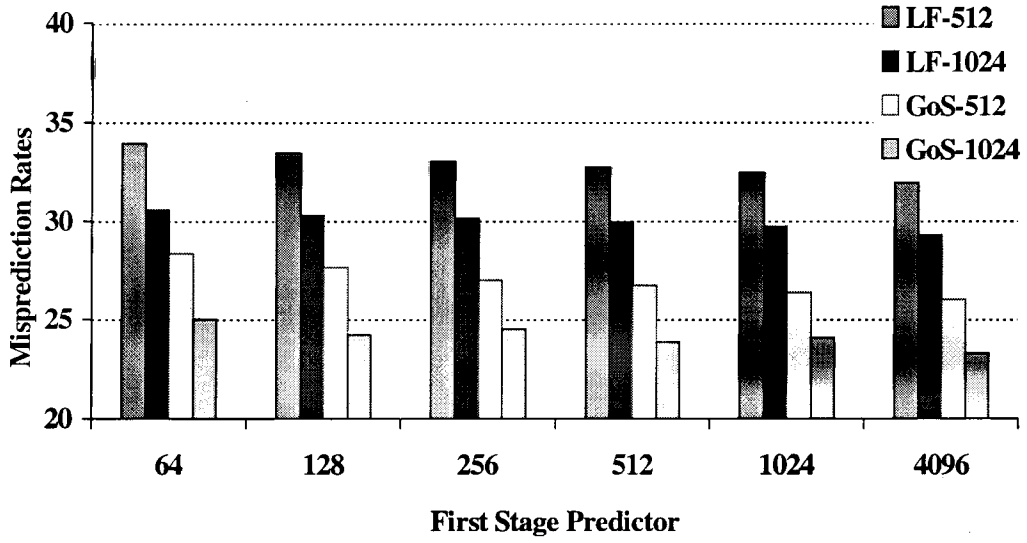
In Table 12, the SPECint95 benchmark programs and C++ programs were used for comparing the indirect misprediction rate between the LF and GoS:

1. The GoS has lower misprediction rates than the LF for most sizes of the BTB (from 64 entries to 4K entries) and the PHT (512entries and 1K entries) for all programs;
2. For the *xlisp* and *perl* program, at 256 entries of BTB and 512 entries of PHT, the GoS improves the misprediction rates at a rate of 44.24% and 15.60% over the LF. For the 512- or 1K-entry PHT table, GoS with 64 entries of BTB works better than the LF with 1K entries of BTB. Therefore, *the GoS yields a misprediction rate better than the LF at less than one-tenth the BTB cost;*
3. For the *gcc* program, at 512 entries of BTB and 512 entries of PHT, the GoS improves the misprediction rate by only 1.36% over the LF. However, at 256 entries or less in BTB and the 1K-entry PHT table, the LF works slightly better than the GoS. This can occur because of the large set of static branches that we discussed in the previous section.

Figure 45 (a) and (b) compare the harmonic mean of misprediction rates for C and C++ programs according to Table 10. There are four predictors namely LF-512 (512-entry PHT table), LF-1024 (1K-entry PHT table), GoS-512 (512-entry PHT), and GoS-1024 (1K-entry PHT table).

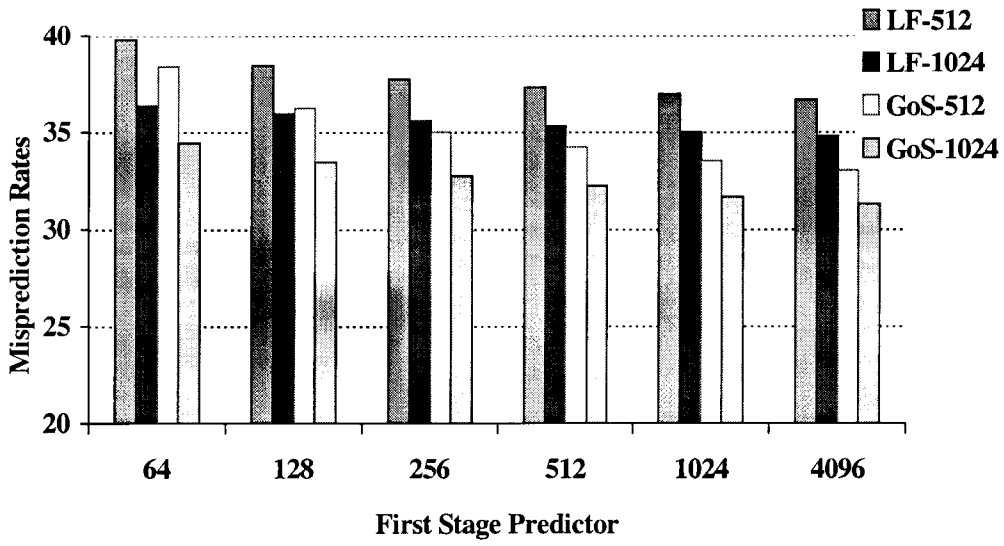
- For C programs in Figure 45(a), the GoS-512 works better than the LF-512 and LF-1024. In addition, the GoS-1024 shows the lowest misprediction rates among other predictors.

- For C++ programs in Figure 45(b), if the BTB has more than 256 entries, the GoS-512 works better than the LF-512 and LF-1024. Otherwise, the LF-1024 works slightly better than the GoS-512. In addition, the GoS-1024 also shows the lowest misprediction rates among other predictors.
- The 256-entry BTB of GoS-512 outperforms the 1-K entry BTB of LF-1024 in both C and C++ programs.
- In general, for most sizes of the BTB-entries, the GoS works better than the LF.

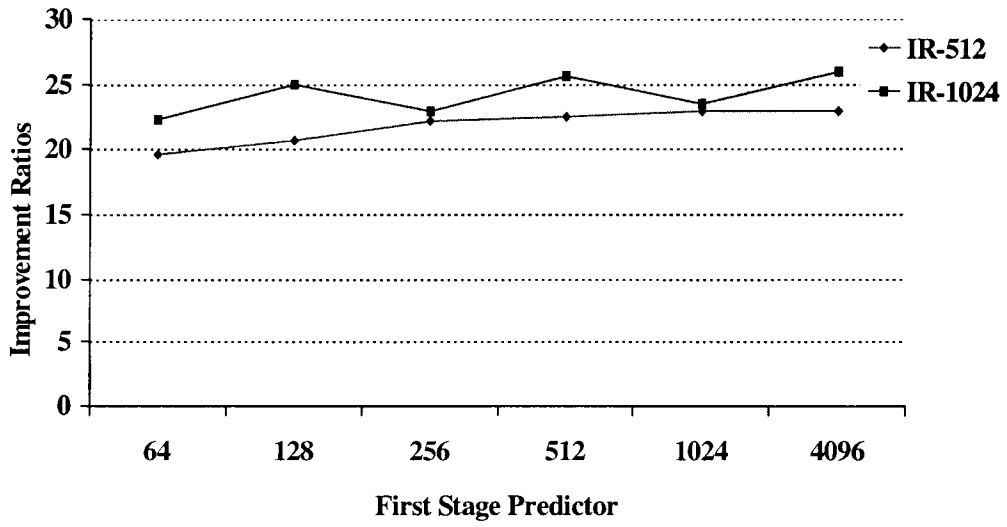


(a) Misprediction Rates for C programs (harmonic mean).

Figure 45. Comparison Misprediction Rates and Improvement Ratios between C and C++ Benchmark programs.

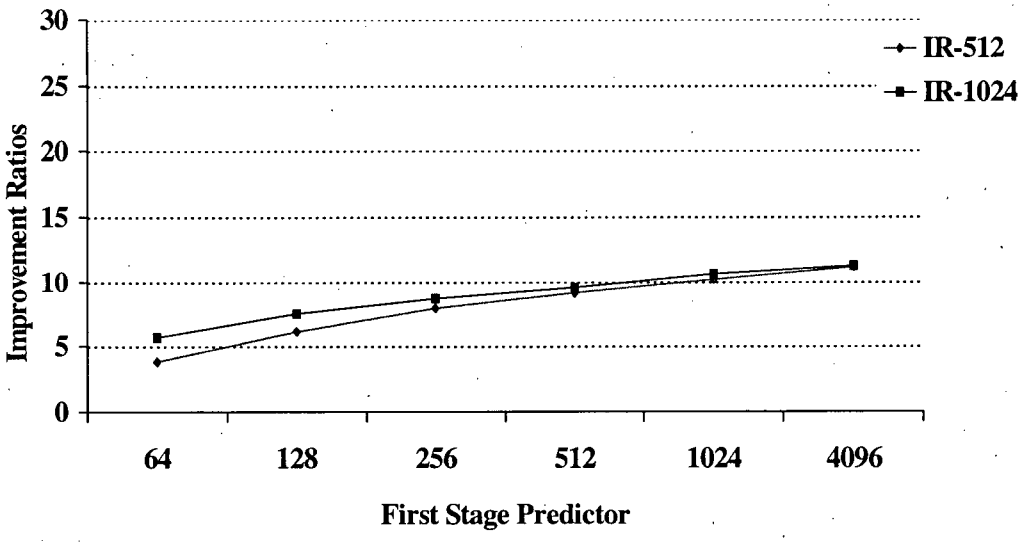


(b) Misprediction Rates for C++ programs (harmonic mean).



(c) Improvement Rates (GoS over LF) for C programs (harmonic mean).

Figure 45. (continued) Comparison Misprediction Rates and Improvement Ratios between C and C++ Benchmark programs.



(d) Improvement Rates (GoS over LF) for C++ programs (harmonic mean).

Figure 45. (continued) Comparison Misprediction Rates and Improvement Ratios between C and C++ Benchmark programs.

Figure 45 shows the improvement ratio (IR) between the LF and GoS according to the sizes of the PHT.

IR-512 =(((MR of the LF-512) – (MR of the GoS-512)) / (MR of the GoS-512)) 100.*

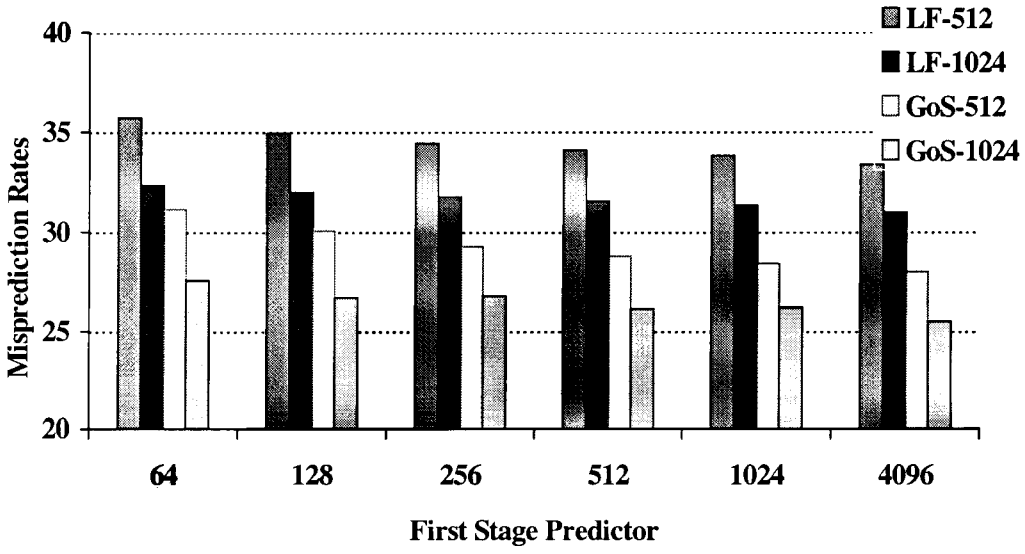
*IR-1024 = (((MR of the LF-1024) – (MR of the GoS-1024)) / (MR of the GoS-1024))*100.*

Where MR represents Misprediction Rates as a harmonic mean of C or C++ programs in Figure 45.

For IR-512, (c) and (d) shows the improved ratios for the C and C++ programs. In the case of C programs, the IR is increased from 19.63% (64-entry of BTB) to 22.93%

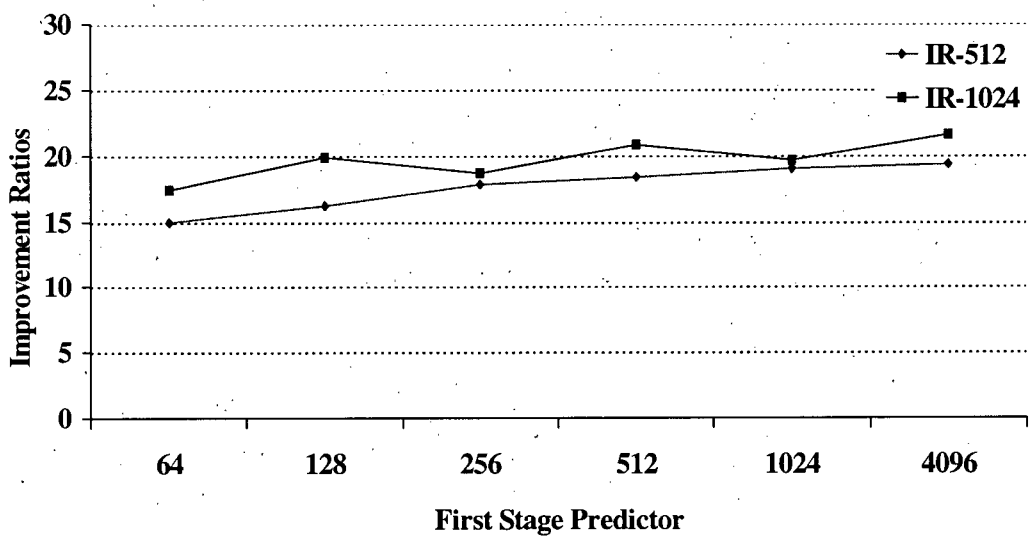
(4096-entry of BTB). However, in the case of C++ programs, the IR is increased by 3.81% (64-entry of BTB) and 11.1% (4096-entry of BTB).

For IR-1024, the IR is slightly higher than the IR-512. In the case of C programs, the IR is increased from 22.31% (64-entry of BTB) to 25.89% (4096-entry of BTB). In the case of C++ programs, the IR is increased by 5.7% (64-entry of BTB) and 11.2% (4096-entry of BTB). In Figure 45(c) and (d), the IR of C programs are higher than C++ because some C programs, such as xlisp and perl, reduce misprediction rates considerably with the GoS compared to the LF. The xlisp and perl are the benchmark programs with high indirect branch density in C programs, while ixx is the only one that has high indirect density in C++ programs [Driesen & Hölzle '98B].



(a) Misprediction Rates for C and C++ programs (harmonic mean).

Figure 46. Comparison Misprediction Rates and Improvement Ratios between the LF and GoS for all Benchmark programs (C and C++ programs, harmonic mean).



(b) Improvement Rates (GoS over LF) for C and C++ programs (harmonic mean).

Figure 46. (continued) Comparison Misprediction Rates and Improvement Ratios between the LF and GoS for all Benchmark programs (C and C++ programs, harmonic mean).

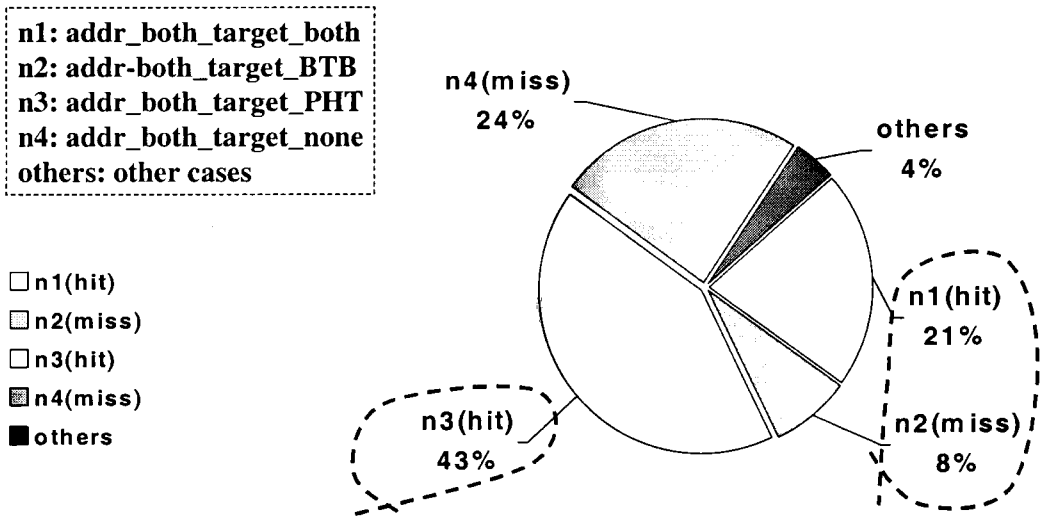
Figure 46 shows misprediction rates as harmonic means for all benchmark programs used. In Figure 46(a), the GoS outperforms the LF for all sizes of both BTB-entries and PHT-entries. Moreover, the GoS-512 can reduce indirect mispredictions better than the LF-1024 for all sizes of BTB.

In Figure 46(b), in the case of the IR-512, the IR increases from 14.9% (64-entry of BTB) to 19.35% (4096-entry of BTB). In the case of IR-1024, the IR increases by 17.41% (64-entry of BTB) and 21.53% (4096-entry of BTB). From the Figure 46, some features of the GoS can be derived:

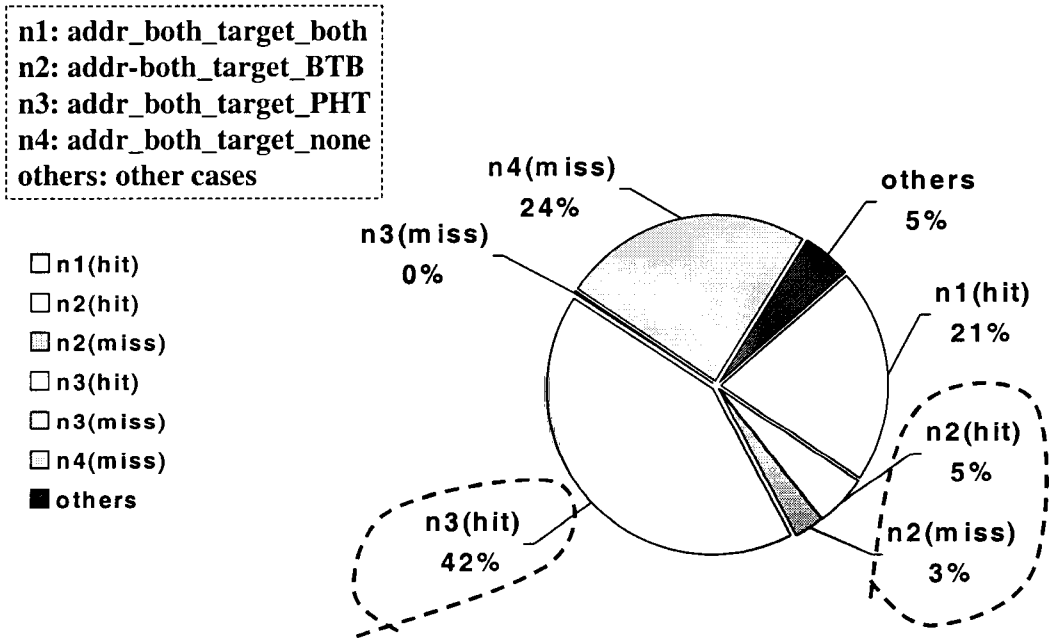
- The GoS reduces indirect mispredictions better than the LF as the indirect density increases. The indirect density can be represented as the inverse of the number of instructions per indirect instruction.
- If the size of the second stage (PHT) is increased, the misprediction rates with the GoS are reduced considerably since the GoS can store more indirect branches with fewer conflicts than the LF.

3.5.5 Analyses of the update rule

In the LF, some mispredictions can be traced to the inefficient update rule. If a predicted target address is wrong, the predictors of both stages are always allowed to update the tables. That means that after resolving an indirect branch the tables of both stages are updated with the new target address. This can remove the possibility for a replaced target being predicted correctly the next time. Therefore, in order to improve the misprediction rate in the LF, the update rule needs to be changed to preserve one of two different targets for the next prediction. This means the target address of one stage (say, the second stage) should be replaced with a new resolved target address while that of the other stage (say, the first stage) should remain for the next time. In order to use the remaining target for the next time, the predict rule needs to be changed in the LF, because if a branch address is found at both stages, then the second stage takes priority of prediction. As we discussed in section 3.3.3, the GoS resolved these problems by using two new mechanisms.

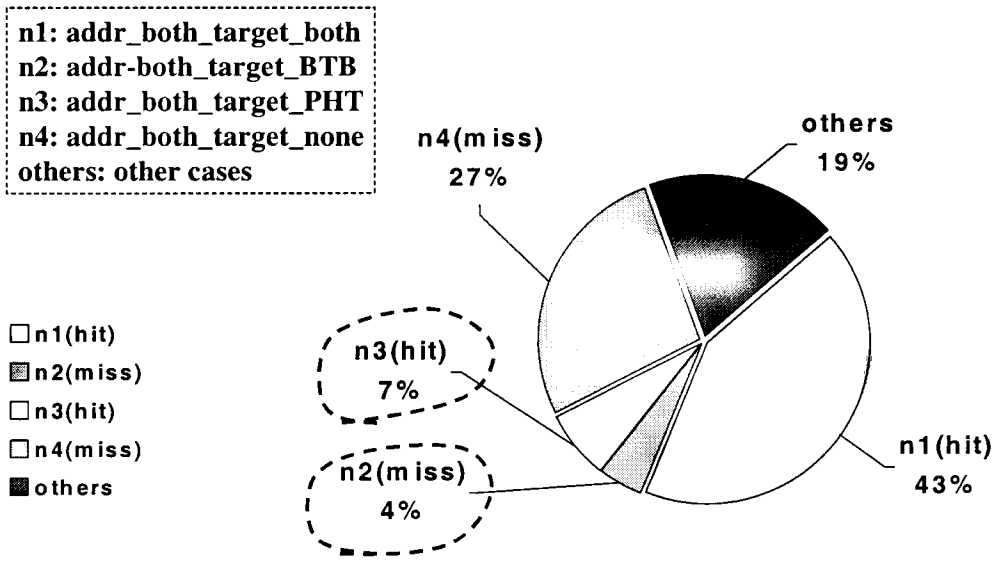


(a) 128-entry filter of LF with 1024-entry PHT (perl, C program)

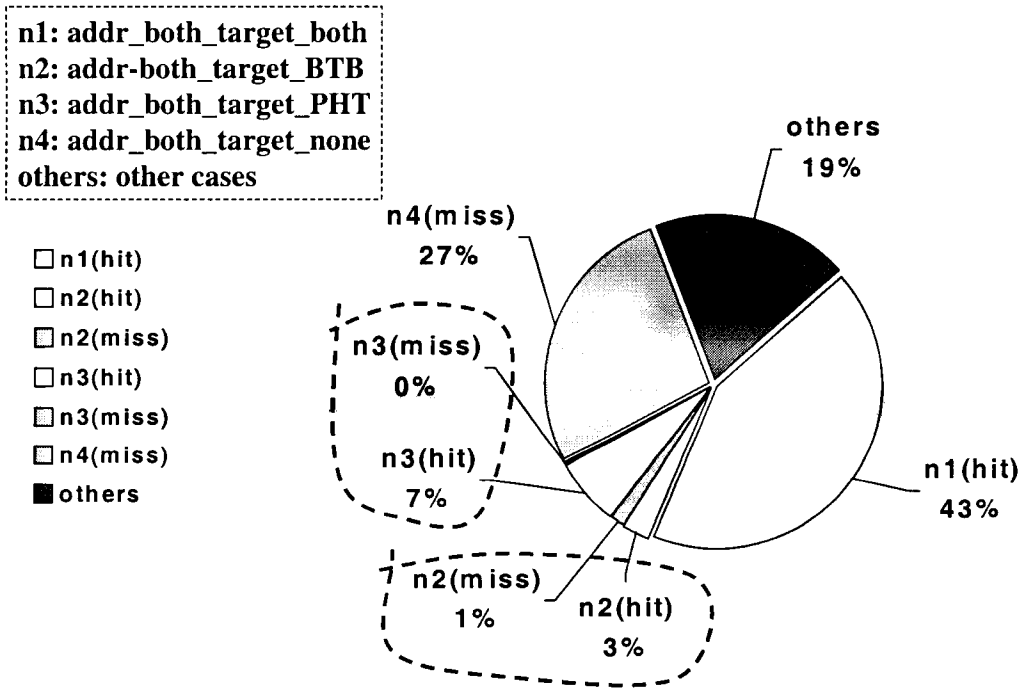


(b) 128-entry filter of GoS with 1024-entry PHT (perl, C program)

Figure 47. Analysis of prediction rates according to cases whether both predictors have a correct target address or not. 'Others' means all other cases except the n1 to n4.



(c) 128-entry filter of LF with 1024-entry PHT (eqn, C++ program)



(d) 128-entry filter of GoS with 1024-entry PHT (eqn, C++ program)

Figure 47. (continued) Analysis of prediction rates according to cases whether both predictors have a correct target address or not. 'Others' means all other cases except the n1 to n4.

There are four cases when both predictors have a prediction:

- **addr_both_target_both (n1)**: Both predictors have the same target and the target is correct. → Correct prediction in both the LF and GoS;
- **addr_both_target_BTB (n2)**: Both predictors have a different target. The first stage has a correct one but the second stage has a wrong one. For the LF, this case leads to a misprediction. But for the GoS, if the second bit of the flag in the first predictor is 0, then this case results in correct prediction. Otherwise, this case is a misprediction. The GoS can reduce misprediction rates considerably by using this predict rule.
- **addr_both_target_PHT (n3)**: Both predictors have a different target. The first stage has a wrong one but the second stage has a correct one. In the LF, this case leads to a correct prediction. Meanwhile, for the GoS, if the flag bit is 0, it leads to a misprediction. However, the possibility for this case is very rare, as little as 1%. Otherwise, it is a correct prediction.
- **addr_both_target_none (n4)**: Both predictors have a target, but neither target is the correct one. This case always leads to a misprediction in both the LF and GoS.

In Figure 47, most mispredictions of the indirect branches occur when two predictors have a simultaneous prediction. Figure 47 shows prediction rates according to the cases from the **n1** to **n4** between the LF and GoS with the 128-entry filter (BTB) and the 1024-entry PHT for 'perl' benchmark program (C program, Figure 47(a) and (b)) and 'eqn' benchmark program (C++ program, Figure 47(c) and (d)). As discussed before, most indirect predictions occur when both stages have a prediction. In Figure 47, 'others' means the prediction rates caused by the cases when one or none of the two stages has a

prediction, which can lead to a hit or miss. However, since these misprediction rates are small compared to other cases, they will be ignored for this section.

The important features provided by Figure 47 are:

- Figure 47(a), for the LF, shows that 96% of the total predictions occur within cases n1 to n4. Among them, even if there is a correct target for n2, the predictions in the LF caused by the n2 always lead to mispredictions because of the inefficient predict rule. The prediction rate caused by the n2 is 8%, which leads to misprediction.
- Figure 47(b), for the GoS, shows that a prediction rate of 95% occurs for case of the n1 to n4. However, the differences between the LF and GoS are the hit and miss rates caused by the cases of the n2 and n3. First, in the GoS, more than half of the predictions (5% out of 8%) lead to a hit instead of a miss; this can improve the misprediction rates by using the GoStay predict and update rule. If the predictions of the n3 lead to a hit in the LF, part of the predictions for the n3 can lead to mispredictions in the GoS. However, since the misprediction rate caused by this case is small (0% in Figure 46(b)), it is possible to disregard the misprediction rates caused by the n3.
- Figure 47(c) shows that 81% of the total predictions occur within case n1 to n4. The prediction rate caused by n2 is 4%, and leads to misprediction.
- Figure 47(d) shows that a prediction rate of 81% occurs in the case of the n1 to n4. In the GoS, more than half of the predictions (3% out of 4%) lead to a hit instead of a miss. The misprediction rates here can be improved by using the GoStay predict and update rule. If the predictions of the n3 lead to the hit in the LF, part of the predictions for the n3 can lead to mispredictions in the GoS. However, since the

misprediction rate caused by this case is small (0% in Figure 47(d)), it is again possible to disregard the misprediction rates caused by the n3.

As we discussed above, the GoS can improve the misprediction rates by selecting the correct target address in the case, n2, by using the GoStay predict rule. Since more than half of the prediction rates in n2 can be changed to a hit instead of a miss. The GoS improved the misprediction rates up to 21.53% compared to the LF for all sizes of the BTB and PHT as shown in Figure 46.

3.6 Chapter conclusions

Due to the increased complexity of application programs, it is quite reasonable to use small functions for code reusability and maintainability. However, these small functions can cause an increase in indirect function calls in object-oriented languages and is a principal cause of indirect branches.

Unfortunately, conventional branch predictors cannot reduce the impact of indirect branch mispredictions since the indirect branch needs the full target address instead of the direction and also the branch target can change with each invocation of the indirect branch. We have discussed several previous indirect branch predictors: the BTB, the Target Cache, and the Cascaded Predictor (with strict filter and with leaky filter).

For the previous indirect two-stage hybrid branch predictors, the leaky filter was found to be the most effective one. However, the accuracy of this predictor is affected by two factors: The conflict misses for small-sized tables (say, less than 512 entries)

considerably increase the misprediction rate. The other factor is the inefficient predict and update rules: For the predict rule, this occurs if the first stage has a correct target address but the second stage has a wrong one, then the assumed priority of the second stage always causes a misprediction. For the update rule, a misprediction can occur when the previous target address is needed after it is updated by a new one, as often happens with indirect branches.

In order to resolve these problems, this chapter has presented a new branch architecture, the GoStay2 predictor, which has two mechanisms that are different from the other hybrid branch predictors. The first mechanism is defined by two functions of a new cache scheme, TAC, employed in the first stage to reduce conflict misses. These functions are the XOR mapping function for indexing the first stage and the BoPLRU replacement policy along with the BSL. The second mechanism is the GoStay predict and update rule to reduce the frequency of wrong predictions caused by inefficient predict and update rules. By using these mechanisms, the GoStay2 reduces the indirect misprediction rate of a 64-entry to 4K-entry BTB (with a 512- or 1K-entry PHT) by 14.9% to 21.53% compared to the best previous indirect branch predictor, the Cascaded predictor (with leaky filter).

Benchmarks	PHT (4-way)	BTB (1st stage predictor)							
		Pred.	n-way	64	128	256	512	1024	4096
First Group: xlisp (C)	512 entries	LF	4	24.14	24.07	24.06	24.05	24.05	24.05
		GoS	**2**	**16.87**	**16.69**	**16.68**	**16.69**	**16.69**	**16.69**
	1024 entries	LF	4	20.95	20.94	20.94	20.94	20.94	20.94
		GoS	**2**	**13.94**	**13.87**	**13.87**	**13.86**	**13.86**	**13.86**
ixx (C++)	512 entries	LF	4	38.18	36.67	36.23	36.11	36.08	36.08
		GoS	**2**	**36.81**	**34.39**	**33.39**	**33.08**	**33.04**	**33.01**
	1024 entries	LF	4	33.86	33.6	33.51	33.49	33.47	33.47
		GoS	**2**	**31.75**	**30.99**	**30.72**	**30.6**	**30.6**	**30.59**
perl (C)	512 entries	LF	4	36.14	35.95	35.71	35.63	35.62	35.63
		GoS	**2**	**33.45**	**32.44**	**30.89**	**30.45**	**30.4**	**30.4**
	1024 entries	LF	4	32.62	32.53	32.43	32.43	32.42	32.42
		GoS	**2**	**30.4**	**27.96**	**30.06**	**27.41**	**30.06**	**27.41**
gcc (C)	512 entries	LF	4	47.94	46.5	45.62	44.65	43.13	39.99
		GoS	**2**	**47.79**	**46.16**	**45.01**	**43.83**	**42.23**	**38.69**
	1024 entries	LF	4	42.72	42.14	41.65	40.96	39.58	36.74
		GoS	**2**	**43.04**	**42.27**	**41.69**	**40.76**	**39.33**	**36.14**
eqn (C++)	512 entries	LF	4	41.56	40.36	39.35	38.59	37.78	37.18
		GoS	**2**	**40.04**	**38.24**	**36.70**	**35.42**	**33.98**	**32.91**
	1024 entries	LF	4	39.25	38.51	37.86	37.29	36.64	36.16
		GoS	**2**	**37.53**	**36.19**	**35.13**	**34.01**	**32.73**	**31.95**
m88ksim (C)	512 entries	LF	4	35.87	34.59	33.68	32.96	32.83	32.76
		GoS	**2**	**32.42**	**31.12**	**29.8**	**29.11**	**28.48**	**28.39**
	1024 entries	LF	4	34.55	33.63	33.09	32.8	32.69	32.65
		GoS	**2**	**31.15**	**29.89**	**29.56**	**29**	**28.44**	**28.21**
Harmonic Mean (Total)	512 entries	LF	4	21.07	20.86	20.72	20.62	20.55	20.44
		GoS	**2**	**19.29**	**18.89**	**18.65**	**18.51**	**18.40**	**18.28**
	1024 entries	LF	4	19.74	19.64	19.56	19.49	19.43	19.32
		GoS	**2**	**17.78**	**17.50**	**17.54**	**17.31**	**17.36**	**17.10**
Second Group: go (C)	512 entries	LF	4	56.1	55.14	53.59	52.58	52.03	52.08
		GoS	**2**	**57.18**	**55.66**	**54.48**	**52.98**	**51.74**	**51.23**
	1024 entries	LF	4	50.41	49.57	48.23	47.16	46.58	46.49
		GoS	**2**	**52.24**	**51.08**	**50.05**	**48.56**	**47.39**	**46.7**
Third Group: deltablue (C++)	512 entries	LF	4	5.87	5.87	5.87	5.87	5.87	5.87
		GoS	**2**	**5.64**	**5.59**	**5.59**	**5.58**	**5.58**	**5.58**
	1024 entries	LF	4	5.73	5.73	5.73	5.73	5.73	5.73
		GoS	**2**	**5.46**	**5.46**	**5.45**	**5.45**	**5.45**	**5.45**

Table 12. Indirect branch misprediction rates according to the BTB entries.

Chapter 4 Conclusions and Future Research

In order to increase the performance of current microprocessor architectures, several techniques are needed to help reduce the memory latency which is caused by the gap between memory and processor performance. These techniques include caching, branch prediction and value prediction, etc.

Since this thesis concentrates on instruction flow, a small-sized on-chip instruction cache memory was considered to improve memory latency. There are three types of cache misses, including compulsory, capacity, and conflict misses. Among these, conflict misses are critical to cache performance and branch penalty for a small-sized on-chip cache memory.

4.1 Conclusions

Since object-oriented languages are widely used, procedure calls have become increasingly used in application programs, causing a significantly increased number of conflict misses in the instruction flow.

This thesis has presented two efficient schemes for improving the HPC: 1) The TAC scheme to reduce conflict misses in the instruction cache memory; and 2) The GoStay2 predictor to reduce indirect branch mispredictions.

4.1.1 Reduction of cache misses

The TAC scheme was designed to reduce instruction cache misses for the frequent procedure calls of object-oriented programs. We discussed several previous cache schemes for reducing conflict misses: direct-mapped, two-way set-associative, four-way set-associative, hash-rehash, victim, and two-way skewed-associative. The victim cache removes many conflict-misses and outperforms a four-way set-associative cache. The two-way skewed-associative cache offers the lowest miss ratio, which is significantly lower than that of a four-way associative cache [Gonzalez et al. '97].

The 2-way skewed-associative cache uses a single flag to avoid conflict misses in bank 0: Each instruction is placed into a bank according to part of its memory address and a flag bit (refer to chapter 2). In general, the efficiency of the 2-way skewed-associative scheme depends on the frequency of conflict misses in bank 0. If conflicts among instructions in bank 0 increase, the efficiency of the 2-way skewed-associative conflict decreases. Therefore, the 2-way skewed-associative works better for traditional programs than object-oriented programs since traditional programs have less procedure calls than object-oriented ones.

The TAC scheme reduces conflict misses effectively by grouping instructions separated by procedure call instructions. There are two steps for removing conflict misses for the TAC scheme:

1. *Initial bank selection*: For each group of instructions separated by a procedure call, the possibility of conflict misses is very rare since each memory address in a group is in sequence (spatial locality). Therefore, if each group is placed into a different bank,

conflict misses can be avoided easily between two adjacent groups (temporal locality).

2. *Final bank selection*: The possibility of conflict misses remains for groups of instructions placed into the same bank after the initial bank selection. In this case, a flag of each cache line in a bank allow a conflict instruction to be placed into the other bank.

With these two bank selection methods, TAC schemes reduce conflict misses better than a 2-way skewed-associative cache. The experimental results in Chapter 2 showed that TAC schemes on a L1 cache (cache sizes: 4 Kbytes to 32 Kbytes, cache line sizes: 8 Bytes to 32 Bytes) improves cache miss rates by up to 9.29% for C programs and 44.44% for C++ programs compared to skewed-associative caches.

Moreover, TAC schemes (on the BTB, 2-way) reduce misprediction rates better than skewed-associative caches (on the BTB, 2-way) by up to 4% for C programs and 6.5% for C++ programs.

4.1.2 Reduction of indirect branch mispredictions

The GoStay2 predictor was designed to reduce indirect branch mispredictions. Several previous branch predictors were discussed for reducing indirect branch mispredictions: BTB, Target Cache, and Cascaded predictor (with strict filter and leaky filter).

The leaky filter, which has two stages, offers the lowest indirect mispredictions. However, this predictor has problems in reducing indirect branch predictions as a result of two factors: 1) conflict misses in the first stage for small tables (fewer than 512

entries); and 2) inefficient predict and update rules. For the predict rule, if the first stage has a correct target address but the second stage has a wrong one, then the assumed priority of the second stage always causes a misprediction. For the update rule, a misprediction can occur when the previous target address is required after it is updated by the new one.

The GoStay2 predictor, which also has two stages, can reduce indirect branch mispredictions by improving these two factors by using two mechanisms:

1. *The first mechanism*: Conflict misses in a small-sized table with less than 512 entries in the first stage can be reduced by using the 2-way TAC scheme instead of a 4-way set-associative.
2. *The second mechanism*: The GoStay predict and update rules (refer to chapter 3) considerably reduce indirect mispredictions caused by inefficient predict and update rules.

With these two mechanisms, GoStay2 predictors reduce indirect mispredictions better than leaky filters. The experimental results in Chapter 3 show that the GoStay2 improves the indirect misprediction rate of a 64-entry to 4K-entry BTB (with a 512- or 1K-entry PHT) by 14.9% to 21.53% compared to a leaky filter.

4.2 Future Research

There are several techniques for reducing memory latency: 1) Cache schemes for reducing cache misses; 2) Control/data flow predictors for instruction level parallelism; etc.

This thesis has focused on designing an efficient instruction cache scheme and indirect branch predictor. Future research could be targeted in three directions such as simulation, caching, and speculation as shown in Figure 48.

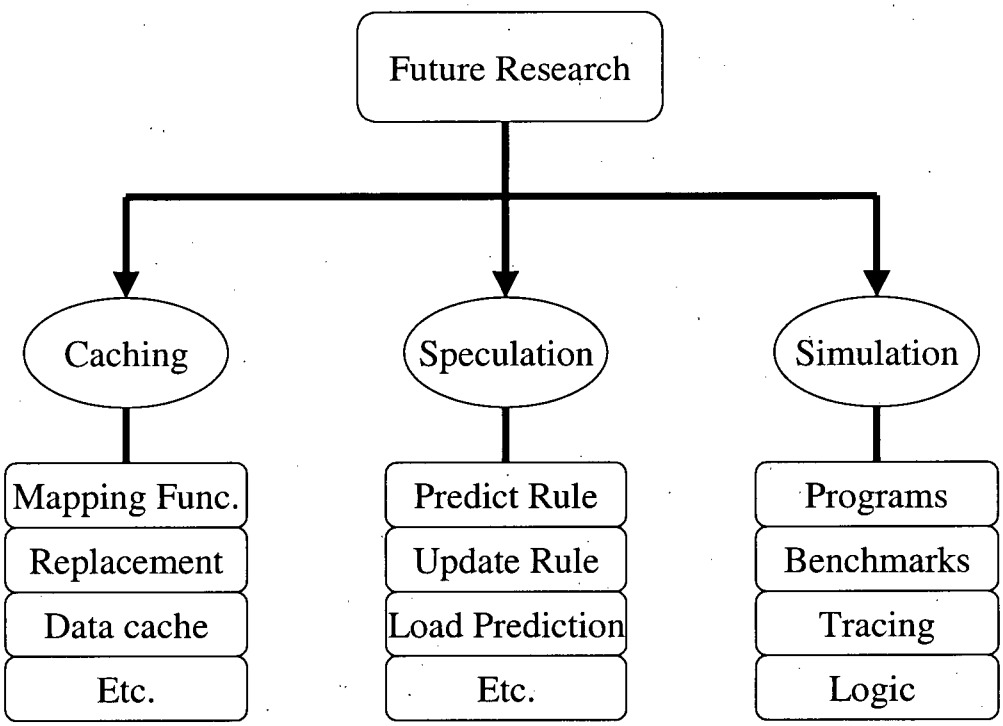


Figure 48. Future Research for caching, speculation, and simulation.

Figure 48 shows three research directions:

1. *Caching*: For a small-sized (less than 32 Kbytes) cache memory, the mapping function is an important factor in reducing conflict misses. Gonzalez et al. ('97) compared the XOR mapping function proposed by Seznec ('93) with the polynomial mapping function proposed by Rau ('91). The polynomial mapping function is based on polynomial arithmetic. For example, an address A =

$< a_{n-1}, \cdots, a_1, a_0 >$ can be considered as a polynomial A (x) = $a_{n-1}x^{n-1}, \cdots, a_1x^1, a_0$, the coefficients of which are in the Galois Field GF (2). For GF (2), all nonzero elements can be represented as 1 and a primitive element *a,* and all coefficients can be implemented as logical AND, and exclusive OR. According to Gonzalez et al. ('97), the polynomial mapping function can reduce conflict misses even better than the XOR mapping function. Therefore, for future research, it will be useful for the TAC scheme to assess the polynomial mapping function instead of the XOR mapping function (refer to chapter 2).

For implementing BoPLRU replacement policy, there can be several ways to use a flag to avoid conflict misses: If the flag is '0', an instruction will be replaced to the other bank on a cache miss; Otherwise, the instruction will be replaced in the current bank. The modification of this replacement policy is also a subject of possible future research.

In the case of data cache memory, data cache misses between traditional and object-oriented programs have no large differences. However, since cache miss rates of the data cache are higher than that of the instruction cache, it is important to future research to design an efficient data cache memory to reduce data cache misses.

2. *Speculation*: High performance computer architectures use aggressive speculation to improve instruction level parallelism. This thesis presented the GoStay2 predictor as a way to avoid stalling the pipeline caused by indirect branch instructions. One of two mechanisms of the GoStay2 predictor is the GoStay

predict and update rule. This GoStay rule can be modified in several ways by changing the usage of the flag and is a subject of future research.

Load instructions, which represent the barrier to data flow, also incur long latencies that can degrade system performance considerably. Fortunately, loads do not fetch random sequences of values. Rather, load instructions often fetch the same values repeatedly, which makes them predictable [Lipasti et al. '96]. A load value predictor can quickly provide a predicted value of the instructions which directly or indirectly consume the load value. Lipasti et al. ('96) introduced the concept of value locality defined as the likelihood of a previously-seen value recurring repeatedly within a storage location. Value locality is visible in many ways [Lipasti et al. '96]:

- Data redundancy: Some programs contain data which has little variation, such as sparse matrix, text files with white spaces, etc;
- Program constants: Sometimes, it is efficient to generate extra code to load program constants from memory into registers;
- Computed branches: In the case of a switch statement, the compiler should generate code to load a register with the base address for the branch;
- Virtual function calls: To call a virtual function call, the compiler should generate code to load a function pointer, which is a run-time value;
- There are many other cases for value locality.

Most recent research has been done on predicting patterns from which values are generated such as stride predictors to keep track of the last value and the previous one [Gonzalez & Gonzalez '98]. Sazeides & Smith ('98) also explores

the use of context predictors that base their prediction on the last of several values seen, thus capturing reference patterns that are not reflected in the simple stride prediction scheme. Much research also has been done on the evaluation of combinations or hybrids of the predictors [Calder et al. '99] [Rychlik et al. '98] [Wang & Franklin '97]. As object-oriented application programs are becoming more popular, efficient load predictors for indirect load values are required to reduce memory latencies. Therefore, one possible future direction beyond this thesis will be in designing an efficient load value predictor.

3. *Simulation*: For this thesis, two simulation programs were used for instruction cache memory and branch predictor with Shade and Spixtools. These programs can be used for future research in the area of: 1) caching and 2) in speculation such as branch predictors. In addition, for implementing load value predictions, there should be three main phases: trace generation, LVP (Load Value Predictor) unit simulation, and microarchitectural simulation such as for the Alpha AXP 21164. For the alpha AXP 21164, traces can be generated with the ATOM tool [Srivastava and Eustace '94].

   For benchmark programs, this thesis used SPECint95 and a suite of C++ programs. However, for future research, SPEC2000 and more C++ programs could be used for evaluating various schemes.

   Finally, H/W implementation (logic) is also needed for all the schemes designed in this thesis and to be developed for future research.

BIBLIOGRAPHY

[1] [Bodin & Seznec '95] F. Bodin, A. Seznec, Skewed-associativity enhances performance predictability, Proceedings of the 22nd International Symposium on Computer Architecture (IEEE-ACM), Santa-Margharita, June 1995 (also IRISA Report No 909).

[2] [Burger & Goodman '97] Doug Burger and James R. Goodman, "Billion-Transistor Architecture", IEEE, Computer, September 1997.

[3] [Buyya '00] Buyya, "High Performance Cluster Computing: Architecture, Systems, and Applications, Conference Tutorial with the 27th ISCA, Vancouver, BC, Canada, June 12-14, 2000.

[4] [Calder & Grunwald '94] B. Calder and D. Grunwald, Reducing indirect function call overhead in c++ programs, in 21st Symposium of Principles of Programming Languages, pages 397-408, 1994.

[5] [Calder et al. '94A] B. Calder, D. Grunwald, and B. Zorn, Quantifying Behavioral Differences Between C and C++ Programs, *Journal of Programming languages*, Vol. 2, No. 4, pp. 313-351, 1994.

[6] [Calder et al. '94B] B. Calder, D. Grunwald, and B. Zorn, Fast & Accurate Instruction Fetch and Branch Prediction, ISCA '94 Conference Proceedings, Chicago, IL. March 1994.

[7] [Calder et al. '99] Brad Calder, Glenn Reinman, and Dean M. Tullsen, Selective Value Prediction, in Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999, pp.64-74.

[8] [Chang et al. '95] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua Chang, Alternative Implementations of Hybrid Branch Predictors, Proceedings of the 28th ACM/IEEE International Symposium on Microarchitecture, Ann Arbor, MI, 1995.

[9] [Chang et al. '97] Po-Yung Chang, Eric Hao, and Yale N. Patt, Target Prediction for Indirect Jumps, Proceedings of the 24th International Symposium on Computer Architecture, Denver, June 1997.

[10] [Chu & Ito '00] Yul Chu and M. R. Ito, The 2-way Thrashing-Avoidance Cache (TAC): An Efficient Instruction Cache Scheme for Object-Oriented Languages, Proceedings of the 17th IEEE International Conference on Computer Design, Austin, Texas, September 2000.

[11] [Cmelik & Keppel '93] R. F. Cmelik and D. Keppel, Shade: A Fast Instruction-Set Simulator for Execution Profiling, Sun Microsystems Laboratories, Technical Report SMLITR-93-12, 1993.

[12] [Diefendroff & Dubey '97] Keith Diefendorff and Pradeep K. Dubey, "How Multimedia Workloads Will Change Processor Design", IEEE, Computer 30(9):43-45, September 1997.

[13] [Driesen & Hölzle '98A] Karel Driesen and Urs Hölzle, Accurate Indirect Branch Prediction, ISCA '98 Conference Proceedings, July 1998.

[14] [Driesen & Hölzle '98B] Karel Driesen and Urs Hölzle, The Cascaded Predictor: Economical and Adaptive Branch Target Prediction, IEEE Micro 31, 1998.

[15] [Fisher '97] Joseph A Fisher, "Walk-Time Techniques: Catalyst for Architecture Change", IEEE, Computer, Sep, 1997.

[16] [Gonzalez & Gonzalez '96] J. Gonzalez and A. Gonzalez, The Potential of Data Value Speculation to Boost ILP, in 12th Annual International Conference on Supercomputing, 1998.

[17] [Gonzalez et al '97] Antonio Gonzalez, Mateo Valero, Nigel Topham, and Joan M. Parcerisa, Eliminating cache conflict misses through XOR-based placement functions,

Proc. Of the ACM international conference on Supercomputing, Vienna (Austria), pp76-83, July 1997.

[18] [Grunwald et al. '98] D. Grunwald, D. Lindsay, and B. Zorn, Static Methods in Hybrid Branch Prediction, in Proceedings of the Intl. Conf. On Parallel Architectures and Compilation Techniques, October 1998.

[19] [Hammond et al. '97] Lance Hammond, Basem Nayfeh, and Kunle Olukotun, "A Single-Chip Multiprocessor", IEEE, Computer, Sep, 1997.

[20] [Handy '93] Jim Handy, The Cache Memory Book, Academic Press, Inc., 1993.

[21] [Hill & Smith '89] M. D. Hill and A. J. Smith, Evaluating Associativity in CPU Caches, IEEE Transactions on Computers, December 1989.

[22] [Hölzle & Ungar '94] Urs Holzle and David Ungar, Do object-oriented languages need special hardware support? Technical Report TRCS 94-21, Department of Computer Science, University of California, Santa Barbara, November 1994.

[23] [Kalamatianos & Kaeli '98] John Kalamatianos and David R. Kaeli, Predicting Indirect Branches via Data Compression, IEEE, MICRO 31, 1998.

[24] [Kozyrakis et al. '97] C. Kozyrakis, S. Perissakis, D. Patterson et al., Scalable Processors in the Billion-Transistor Era:IRAM, IEEE, Computer, vol 30, no. 9, September 1997, p75-78.

[25] [Lipasti & Shen '97] Mikko H. Lipasti and John Paul Shen, Superspeculative Microarchitecture for Beyond AD2000, IEEE, Computer, September 1997.

[26] [Lipasti et al. '96] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, Value locality and load value prediction, In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), October 1996.

[27] [McFarling '93] S. McFarling, Combining branch predictors, Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[28] [Mills '96] Jack Mills, An industrial perspective on computer architecture, ACM Computing Survey, Volume28, 4es, 1996.

[29] [Mudge '96] Trevor Mudge, Strategic directions In Computer Architecture, ACM Computing Surveys, Volume28, Issue4, 1996.

[30] [Nair '95] Ravi Nair, Path-Based Branch Correlation, Proceedings of MICRO-28, 1995.

[31] [Olsen '96] Dan Olsen, Computational Resources and the Internet, ACM Computing Survey, Volume28, 4es, 1996.

[32] [Pasquale '96] Joseph Pasquale, Towards Internet Computing, ACM Computing Survey, V28, 4e, 1996.

[33] [Patt et al. '97] Yale N.Patt, S. J. Patel, M. Evers, D. H. Friendly, J. Stark, One Billion Transistors, One Uniprocessor, One chip, IEEE, Computer, September 1997.

[34] [Radhakrishnan & John '98] R. Radhakrishnan and L. John, Execution Characteristics of Object-oriented Programs on the UltraSPARC-II, Proceedings of the 5th *Int. Conf. on High Performance Computing*, Dec. 1998.

[35] [Rau '91] B. R. Rau, Pseudo-Randomly Interleaved Memories, in Proc. Int. Symp. on Computer Architecture, 1991, pp. 74-83.

[36] [Rychlik et al. '98] Bhuslav Rychlik, John Faistl, Bryon Krug, and John Paul Shen, Eficacy and Performance Impact of Value Prediction, Technical Report CMuART-1998-04.

[37] [Seznec '93] A. Seznec, A case for two-way skewed associative caches, Proc. of the 20th *Int. Symp. on Computer Architecture*, May 1993, pp 169-178.

[38] [Seznec '97] Andre Seznec, A new case for Skewed-Associativity, IRISA Report No. 1114, July 1997.

[39] [Srivastava & Eustace '94] Amitabh Srivastava and Alan Eustace, ATOM: A system for building customized program analysis tools. In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, 1994, pp. 196-205.

[40] [Tullsen & Seng '99] Dean M. Tullsen and John S. Seng, Storageless Value Prediction using Prior Register Values, in Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999, pp.270-279.

[41] [Waingold et al. '97] Elliot Waingold et.al., "Baring It All to Software: Raw Machines", IEEE, Computer, Sep, 1997.

[42] [Wang & Franklin '97] Kai Wang and Manoj Franklin, Highly Accurate Data Value Prediction using Hybrid Predictors, in Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, 1997, pp. 281-290.

[43] [Yeh & Patt '93] Tse-Yu Yeh and Yale N. Patt, A comparison of dynamic branch predictors that use two levels of branch history, ISCA, pages 257-266, 1993.

Appendix A Experimental results for TAC schemes

A.1 In the case of 8 bytes of cache line size

A.1.1 Cache size: 4 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	13.5741	12.3575	12.0427	11.7404	11.404	11.2711
m88ksim	9.9471	7.4542	6.3708	3.7289	5.1271	3.5799
compress	0.1688	0.1277	0.1209	0.0746	0.0823	0.074
li	5.1875	1.8762	0.3607	0.093	0.3886	0.1612
C++ Programs						
deltablue	8.0334	7.7241	6.6252	6.8178	5.1844	5.2465
ixx	11.1933	9.8344	6.0157	4.1165	5.451	5.3853
eqn	8.9639	6.558	5.4569	3.3671	4.7608	4.2074
C harmonic mean	7.22	5.45	4.72	3.91	4.25	3.77
C++ harmonic mean	9.40	8.04	6.03	4.77	5.13	4.95

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	11.1119	11.1267	2.62871	10.646	10.8381	5.871689
m88ksim	4.7821	4.9144	7.2144	3.3723	3.5435	6.156036
compress	0.0777	0.0785	5.92021	0.0574	0.0598	28.91986
li	0.2883	0.2863	34.7901	0.1421	0.1447	13.44124
C++ Programs						
deltablue	4.6123	4.5402	12.4038	4.986	4.9716	5.224629
ixx	5.2906	5.2923	3.03179	4.1889	4.5289	28.5612
eqn	4.5504	4.4957	4.62377	3.7011	3.7056	13.67972
C harmonic mean			5.58			9.06
C++ harmonic mean			4.79			10.02

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.1.2 Cache size: 8 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	9.1722	7.7615	6.2134	4.9915	5.9139	5.0517
m88ksim	5.8685	4.1365	2.1018	1.4363	2.3495	1.6656
compress	0.087	0.0701	0.0283	0.0175	0.0311	0.0195
li	0.7856	0.6061	0.1168	0.0079	0.0305	0.0132
C++ Programs						
deltablue	4.7708	2.7842	1.6601	0.0861	1.5225	0.0117
ixx	6.955	3.5307	1.4447	0.2425	1.114	0.3466
eqn	5.6203	2.7741	1.4158	0.8877	1.8375	1.2184
C harmonic mean	3.98	3.14	2.12	1.61	2.08	1.69
C++ harmonic mean	5.78	3.03	1.51	0.41	1.49	0.53

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	5.6966	5.6987	3.81456	4.894	5.0781	3.222313
m88ksim	2.163	2.1757	8.62228	1.5623	1.6339	6.612046
compress	0.0265	0.0277	17.3585	0.0183	0.0193	6.557377
li	0.0145	0.0131	110.345	0.0119	0.013	10.92437
C++ Programs						
deltablue	1.0737	0.9787	41.7994	0.0108	0.0108	8.333333
ixx	0.8022	0.8011	38.8681	0.2491	0.2873	39.14091
eqn	1.732	1.7324	6.09122	1.195	1.1965	1.958159
C harmonic mean			8.99			5.67
C++ harmonic mean			14.03			4.57

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.1.3 Cache size: 16 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	6.1155	4.2874	3.6041	3.1364	3.5366	3.2786
m88ksim	3.5999	1.6063	1.1218	0.777	1.0883	0.977
compress	0.0599	0.0242	0.0171	0.0109	0.0153	0.0126
li	0.4797	0.0682	0.0053	0.0036	0.0055	0.0048
C++ Programs						
deltablue	2.5623	0.9306	0.2434	0.0451	0.513	0.005
ixx	3.6767	1.8135	0.1891	0.0183	0.2015	0.0217
eqn	2.9243	1.0045	0.5393	0.4198	0.6108	0.5092
C harmonic mean	2.56	1.50	1.19	0.98	1.16	1.07
C++ harmonic mean	3.05	1.25	0.32	0.16	0.44	0.18

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	3.3943	3.393	4.19232	3.1665	3.2477	3.540186
m88ksim	1.0791	1.076	0.85256	0.8832	0.9376	10.62047
compress	0.0144	0.0148	6.25	0.0121	0.0125	4.132231
li	0.0051	0.0051	7.84314	0.0045	0.0049	6.666667
C++ Programs						
deltablue	0.1028	0.1389	399.027	0.0049	0.0048	2.040816
ixx	0.0661	0.0659	204.841	0.0214	0.0253	1.401869
eqn	0.5237	0.5245	16.6317	0.4615	0.4615	10.33586
C harmonic mean			2.35			5.20
C++ harmonic mean			44.44			2.31

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.1.4 Cache size: 32 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	3.4646	2.4058	2.011	1.7609	1.9741	1.9271
m88ksim	1.8787	1.0754	0.7072	0.6892	0.7964	0.7362
compress	0.0276	0.0163	0.0102	0.0101	0.0114	0.0104
li	0.0225	0.0046	0.0031	0.003	0.0037	0.0032
C++ Programs						
deltablue	1.7674	0.3336	0.0452	0.0444	0.0589	0.0044
ixx	1.6477	0.4922	0.0557	0.0159	0.0355	0.0163
eqn	1.7602	0.3163	0.1373	0.0532	0.1274	0.051
C harmonic mean	1.35	0.88	0.68	0.62	0.70	0.67
C++ harmonic mean	1.73	0.38	0.08	0.04	0.07	0.02

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	1.9278	1.9277	2.4017	1.7801	1.8437	8.257963
m88ksim	0.7962	0.8064	0.02512	0.7131	0.7253	3.239377
compress	0.0113	0.0114	0.88496	0.0102	0.0103	1.960784
li	0.0036	0.0035	2.77778	0.0031	0.0032	3.225806
C++ Programs						
deltablue	0.0479	0.0489	22.9645	0.0043	0.0044	2.325581
ixx	0.0185	0.018	91.8919	0.0162	0.0163	0.617284
eqn	0.0989	0.1002	28.817	0.0497	0.0507	2.615694
C harmonic mean			0.10			3.20
C++ harmonic mean			33.66			1.23

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.2 In the case of 16 bytes of cache line size

A.2.1 Cache size: 4 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	8.6789	7.9353	7.946	8.0305	7.4845	7.698
m88ksim	6.3876	5.0672	4.4541	3.3018	4.1036	2.9957
compress	0.1028	0.0795	0.0768	0.0568	0.0637	0.0475
li	3.3711	1.2779	0.3358	0.0841	0.3675	0.1499
C++ Programs						
deltablue	5.116	4.7752	4.2907	4.5764	4.0781	4.149
ixx	7.5845	6.8999	4.6471	3.3742	4.5208	4.2719
eqn	5.9877	4.8258	4.4649	2.7172	4.0775	3.2816
C harmonic mean	4.64	3.59	3.20	2.87	3.00	2.72
C++ harmonic mean	6.23	5.50	4.47	3.56	4.23	3.90

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	7.393	7.4022	1.237657	7.3405	7.4604	4.87024
m88ksim	3.7967	3.7306	8.083336	2.7533	2.8557	8.803981
compress	0.0599	0.0592	6.343907	0.0406	0.0435	16.99507
li	0.1789	0.1798	105.422	0.1233	0.1271	21.5734
C++ Programs						
deltablue	3.07731	3.7096	32.52159	3.7354	3.7374	11.07244
ixx	4.1548	4.1553	8.809088	3.3772	3.6033	26.49236
eqn	3.8129	3.8116	6.9396	2.9355	2.9405	11.79015
C harmonic mean			3.64			9.43
C++ harmonic mean			10.40			14.09

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.2.2 Cache size: 8 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	5.9347	5.0115	4.1727	3.4524	4.1238	3.5264
m88ksim	3.8189	2.8224	1.5402	0.9474	1.3996	0.8247
compress	0.0564	0.0475	0.021	0.011	0.0173	0.0122
li	0.5394	0.4232	0.0834	0.0052	0.0238	0.0099
C++ Programs						
deltablue	3.0746	1.9852	1.3405	0.2427	1.0326	0.0076
ixx	4.7679	2.5423	1.3825	0.2884	1.1473	0.3752
eqn	3.879	2.0957	1.1382	0.6186	1.1265	0.8662
C harmonic mean	2.59	2.08	1.45	1.10	1.39	1.09
C++ harmonic mean	3.91	2.21	1.29	0.38	1.10	0.42

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	3.9645	3.9744	4.018161	3.3503	3.483	5.256246
m88ksim	1.3202	1.2846	6.01424	0.7526	0.7918	9.580122
compress	0.0163	0.0159	6.134969	0.0114	0.0118	7.017544
li	0.0106	0.0094	124.5283	0.0088	0.0094	12.5
C++ Programs						
deltablue	0.6488	0.6002	59.15536	0.0062	0.0087	22.58065
ixx	0.9444	0.9431	21.48454	0.2642	0.2986	42.01363
eqn	1.034	1.1002	8.945841	0.8192	0.823	5.737305
C harmonic mean			6.82			7.73
C++ harmonic mean			17.12			12.38

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.2.3 Cache size: 16 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	3.9905	2.7931	2.3425	1.9864	2.2363	2.0591
m88ksim	2.4346	1.1673	0.8468	0.4722	0.7035	0.4043
compress	0.0399	0.0176	0.0133	0.0064	0.0087	0.0075
li	0.2987	0.0618	0.0039	0.0022	0.004	0.0029
C++ Programs						
deltablue	1.6494	0.6624	0.2994	0.0254	0.2398	0.0029
ixx	1.6494	1.2925	0.1574	0.0116	0.138	0.0145
eqn	1.6494	0.8677	0.3923	0.2721	0.3947	0.338
C harmonic mean	1.69	1.01	0.80	0.62	0.74	0.62
C++ harmonic mean	1.65	0.94	0.28	0.10	0.26	0.12

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	2.1576	2.1587	3.647571	1.9997	2.0545	2.970446
m88ksim	0.6799	0.6731	3.471099	0.3636	0.3926	11.19362
compress	0.0084	0.0086	3.571429	0.0071	0.0075	5.633803
li	0.0033	0.0033	21.21212	0.0028	0.0029	3.571429
C++ Programs						
deltablue	0.1725	0.1872	39.01449	0.0028	0.0029	3.571429
ixx	0.0416	0.0375	231.7308	0.0144	0.0166	0.694444
eqn	0.3592	0.3507	9.883073	0.3127	0.3143	8.090822
C harmonic mean			4.50			4.53
C++ harmonic mean			22.88			1.63

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.2.4 Cache size: 32 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	2.2738	1.5641	1.3168	1.1609	1.3269	1.2732
m88ksim	1.3058	0.7318	0.4034	0.3894	0.4665	0.2609
compress	0.0191	0.0111	0.0057	0.0057	0.0063	0.0059
li	0.0141	0.0031	0.0017	0.0016	0.0021	0.0019
C++ Programs						
deltablue	1.1016	0.2278	0.0252	0.0247	0.0298	0.0025
ixx	1.0529	0.3239	0.0405	0.0087	0.0137	0.0091
eqn	1.2675	0.2233	0.098	0.0428	0.0855	0.0428
C harmonic mean	0.90	0.58	0.43	0.39	0.45	0.39
C++ harmonic mean	1.14	0.26	0.05	0.03	0.04	0.02

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	1.2864	1.2861	3.148321	1.1738	1.2163	8.468223
m88ksim	0.4583	0.4659	1.789221	0.2516	0.2546	3.696343
compress	0.0062	0.0062	1.612903	0.0057	0.0057	3.508772
li	0.002	0.002	5	0.0018	0.0018	5.555556
C++ Programs						
deltablue	0.0271	0.03	9.9631	0.0024	0.0024	4.166667
ixx	0.0112	0.0109	22.32143	0.009	0.0091	1.111111
eqn	0.0711	0.0709	20.25316	0.0419	0.0426	2.147971
C harmonic mean			2.36			4.69
C++ harmonic mean			15.42			1.87

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.3 In the case of 32 bytes of cache line size

A.3.1 Cache size: 4 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	7.9949	5.4971	5.5726	5.6542	5.4127	5.5763
m88ksim	4.5646	4.0027	3.6464	3.7047	3.2517	2.9957
compress	0.071	0.0598	0.0577	0.0603	0.0494	0.046
li	1.9055	0.9577	0.3082	0.0896	0.2095	0.2128
C++ Programs						
deltablue	3.5258	3.2252	3.0665	3.152	2.9576	3.0722
ixx	5.4656	4.9618	4.0429	3.1098	3.8383	3.696
eqn	4.4088	3.9352	4.0752	2.9875	3.8065	2.7779
C harmonic mean	3.63	2.63	2.40	2.38	2.23	2.21
C++ harmonic mean	4.47	4.04	3.73	3.08	3.53	3.18

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	5.3645	5.3725	0.898499	5.3663	5.4436	3.913311
m88ksim	3.1451	3.1806	3.389399	2.7533	2.8557	8.803981
compress	0.0474	0.0474	4.219409	0.0416	0.0429	10.57692
li	0.1761	0.1798	18.9665	0.1338	0.1493	59.04335
C++ Programs						
deltablue	2.7039	2.6597	9.382743	2.8521	2.8392	7.717121
ixx	3.4364	3.4823	11.69538	3.054	3.238	21.02161
eqn	3.5743	3.5746	6.487439	2.5455	2.5454	9.129837
C harmonic mean			2.36			8.32
C++ harmonic mean			8.66			10.46

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.3.2 Cache size: 8 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	4.1804	3.6679	3.1195	2.7556	3.08	2.833
m88ksim	2.8292	2.2555	1.3272	0.723	1.3233	0.8247
compress	0.04	0.0369	0.0198	0.0073	0.0187	0.0082
li	0.3859	0.3035	0.1161	0.0039	0.0235	0.0111
C++ Programs						
deltablue	2.1809	1.5769	1.2615	0.5718	1.1811	0.0543
ixx	3.5475	2.052	1.2953	0.4001	0.8825	0.5443
eqn	2.8563	1.6637	1.0094	0.5411	1.0382	0.7688
C harmonic mean	1.86	1.57	1.15	0.87	1.11	0.92
C++ harmonic mean	2.86	1.76	1.19	0.50	1.03	0.46

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	3.0053	3.0114	2.485609	2.6558	2.7596	6.672189
m88ksim	1.3065	1.3048	1.285878	0.7526	0.7918	9.580122
compress	0.0163	0.0168	14.72393	0.0078	0.008	5.128205
li	0.0088	0.0087	167.0455	0.009	0.0097	23.33333
C++ Programs						
deltablue	0.9927	1.004	18.97854	0.0527	0.0437	3.036053
ixx	0.733	0.7324	20.39563	0.3895	0.4358	39.74326
eqn	0.9214	0.9343	12.67636	0.7089	0.7109	8.449711
C harmonic mean			3.19			8.13
C++ harmonic mean			16.61			6.34

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.3.3 Cache size: 16 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	2.8254	1.9897	1.6636	1.3693	1.6726	1.3945
m88ksim	1.8028	0.877	0.6775	0.3243	0.4904	0.4043
compress	0.0285	0.0128	0.0107	0.0041	0.0071	0.0048
li	0.1997	0.0503	0.0193	0.0017	0.0057	0.0021
C++ Programs						
deltablue	1.2023	0.6767	0.2733	0.0151	0.1851	0.0019
ixx	1.8822	1.0502	0.1395	0.0105	0.2141	0.0146
eqn	0.9408	0.1709	0.0773	0.0401	0.2919	0.2378
C harmonic mean	1.21	0.73	0.59	0.42	0.54	0.45
C++ harmonic mean	1.34	0.63	0.16	0.02	0.23	0.08

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	1.6061	1.606	4.140464	1.3696	1.4071	1.818049
m88ksim	0.4589	0.4575	6.864241	0.3036	0.3926	33.16864
compress	0.0055	0.0056	29.09091	0.0046	0.0048	4.347826
li	0.0027	0.0026	111.1111	0.0019	0.0021	10.52632
C++ Programs						
deltablue	0.0599	0.0596	209.015	0.0018	0.0018	5.555556
ixx	0.0797	0.0822	168.6324	0.0133	0.0147	9.774436
eqn	0.2618	0.2644	11.49733	0.2253	0.2261	5.548158
C harmonic mean			9.29			4.42
C++ harmonic mean			30.71			6.49

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.

A.3.4 Cache size: 32 Kbytes

Benchmark Programs	Cache schemes					
	Direct-mapped	2-way set-associative	4-way set-associative	16-way set-associative	2-way skew	4-way skew
SPECint95 (C programs)						
gcc	1.6247	1.0923	0.9306	0.8345	0.9405	0.9049
m88ksim	0.8457	0.5073	0.2432	0.2295	0.3007	0.2609
compress	0.0124	0.0076	0.0033	0.0033	0.0039	0.0034
li	0.0188	0.0027	0.0011	0.001	0.0018	0.0011
C++ Programs						
deltablue	0.8077	0.1986	0.0147	0.0138	0.0162	0.0014
ixx	0.7429	0.2627	0.0343	0.005	0.0128	0.0052
eqn	0.9408	0.1709	0.0773	0.0401	0.0774	0.0423
C harmonic mean	0.63	0.40	0.29	0.27	0.31	0.29
C++ harmonic mean	0.83	0.21	0.04	0.02	0.04	0.02

Benchmark Programs	Cache schemes					
	2-way TAC scheme			4-way TAC scheme		
	2-bit counter	3-bit counter	IR of TAC over skew	2-bit counter	3-bit counter	IR of TAC over skew
SPECint95 (C programs)						
gcc	0.9064	0.9068	3.762136	0.8434	0.8699	7.291914
m88ksim	0.2943	0.2983	2.174652	0.2516	0.2546	3.696343
compress	0.0038	0.0038	2.631579	0.0033	0.0034	3.030303
li	0.0015	0.0013	20	0.001	0.0011	10
C++ Programs						
deltablue	0.0151	0.015	7.284768	0.00135	0.0014	3.703704
ixx	0.0065	0.0064	96.92308	0.005	0.0054	4
eqn	0.0637	0.063	21.50706	0.0377	0.0379	12.20159
C harmonic mean			3.46			4.78
C++ harmonic mean			15.46			4.98

** IR: Improvement Ratios (Refer to p67 in Chapter 2).
** IR of TAC (2-way and 4-way) over skew uses '2-bit counter'.