# An Efficient Instruction Cache Scheme for Object-Oriented Languages

Yul Chu and M. R. Ito
Electrical and Computer Engineering Department, University of British Columbia
2356 Main Mall, Vancouver, BC V6T1Z4, Canada
{yulc, mito} @ece.ubc.ca

**Abstract**

*In this paper, we present an efficient cache scheme, which can considerably reduce instruction cache misses caused by procedure call/returns. This scheme employs N-way banks and XOR mapping functions. The main function of this scheme 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. We have developed a simulation program by using Shade and Spixtools, provided by SUN Microsystems, on an ultra SPARC/10 processor. Our experimental results show that these schemes reduce conflict misses more effectively than skewed-associative caches in both C (up to 9.29% improvement) and C++ (up to 30.71% improvement) programs on L1 caches. In addition, they also allow for a significant miss reduction on Branch Target Buffers (BTB).*

## 1. Introduction

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

Previous research has shown that instruction cache misses are one of the most critical factors in degrading system performance for object-oriented application programs. Calder et al (’94) showed that object-oriented (C++) programs 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 of larger cache blocks (C++: C = 8.0: 4.9), 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++, suffer from instruction cache conflicts.

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. However, Calder et al (’94) and Holzle & Ungar (’94) observed that the data cache misses for both programs were seen to be similar. So we 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]. In this paper, we present a new cache scheme, which can effectively reduce instruction cache misses caused by call/returns.

This paper is organized as follows: Section 2 explains cache misses and skewed-associative caches; section 3 presents a new instruction cache scheme; section 4 describes simulation methodology and benchmark programs; section 5 presents our simulation results; and section 6 provides our conclusions.

## 2. Cache Misses

### 2.1 Total cache miss vs. Conflict Miss Ratios

Gonzalez et al (’97) generated the miss ratios for several cache schemes as shown in Figure 1: 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 1 by using the SPEC95 benchmark suite by implementing a cache memory (8 kilobytes capacity and 32 bytes per line).

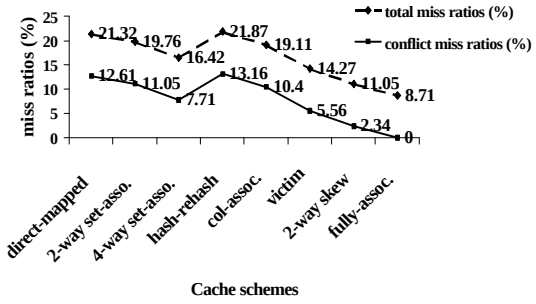



**Figure 1.** Cache Miss Ratios (%) of several cache schemes

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 1. 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 particular scheme – total miss ratio for the fully-associative scheme).

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].

## 2.2 Skewed-associative caches

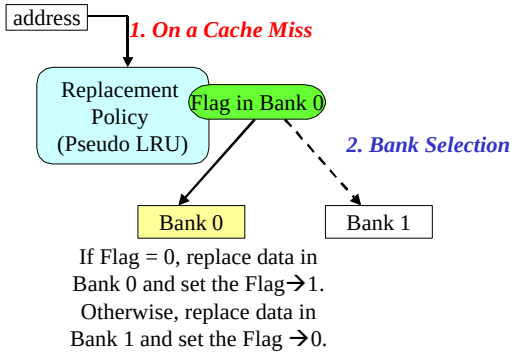



**Figure 2.** Bank selection of a 2-way skewed-associative cache.

Skewed associative caches have been previously proposed [Seznec '97]. An N-way skewed-associative cache consists of N distinct banks that are accessed simultaneously with different mapping functions. Figure 2 shows a 2-way skewed-associative cache using a pseudo-LRU replacement policy by associating a one-bit flag to each line in bank 0 on a cache miss.

Bodin & Seznec ('95) presented skewing functions that are obtained by XORing a few bits in the address of a memory block.

# 3. N-Way Thrashing-Avoidance Cache (TAC)

There are two main reasons why we need to design a new instruction cache memory:

- As technology changes, smaller on-chip 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, we need a new cache memory scheme to reduce instruction cache misses and focus on reducing thrashing conflict misses (i.e., a commonly used location is displaced by another commonly used location in a cycle).

## 3.1 An Overview of an N-Way TAC Scheme

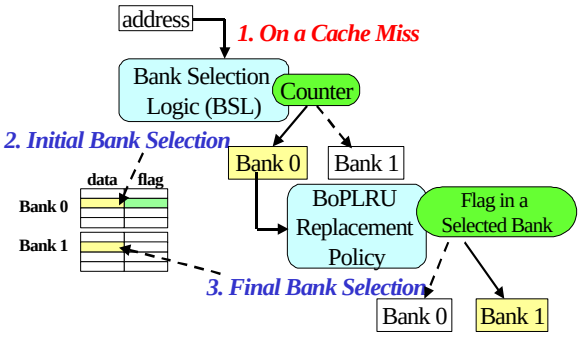



**Figure 3.** The basic operation of a 2-way TAC scheme

An N-way TAC scheme is built with N distinct banks. Since Gonzales et al ('97) showed that XOR mapping functions work well for reducing conflict misses; the N-way TAC employs XOR mapping functions for accessing the instruction cache memory.

On a cache miss, the initial bank selection mechanism selects a bank according to the BSL (Bank Selection Logic) and the final bank selection mechanism determines the bank to update according to the BoPLRU (Bank-originated Pseudo LRU replacement policy) replacement policy (Figure 3).

Each cache line in a bank 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, we represent a cache line of a TAC scheme as just a data and flag throughout this paper and omit the tag part.

### 3.2 Initial bank selection mechanism

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 a 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. The n most significant bits of the x-bit counter represent the selected bank for each instruction. Each bank can be selected on every $2^{x-n}$ procedural calls. For example, if x = 2 and n = 1, then there are two banks ($\mathbf{2}^n$ = 2) and a bank is switched 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.

Figure 4 shows how a 2-bit counter (x = 2 and n = 1) in the BSL works with the flow of example instructions. In Figure 4(a), 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.

The detailed operations of the 2-bit counter in the BSL, in Figure 4(b), 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) and so on.

In Figure 4(c), 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.

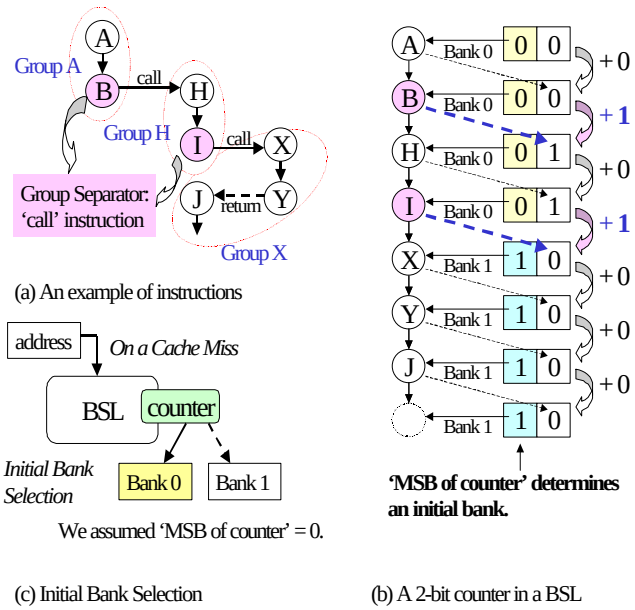



**Figure 4.** The operation of the BSL according to a flow of instructions (2-bit counter).

### 3.3 Final bank selection mechanism

After a bank is initially selected, the BoPLRU (Bank-originated Pseudo LRU replacement policy) determines the final bank for updating a line as a correction mechanism by checking the flag for the selected cache line.

Figure 5 shows the Pseudo code for the BoPLRU. 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, we expect any conflicting data remains in a bank safely for a while.

Figure 5 also shows the pseudo code for the BoPLRU replacement policy for the N-way ($N = 2^n$) TAC scheme [Chu '00].

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.

As an example, the BoPLRU operation of the 1-bit flag, 2-way TAC scheme, is shown in Figure 6: We assumed that the BSL initially selects 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.

```
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 5**. Pseudo code for the BoPLRU Replacement Policy

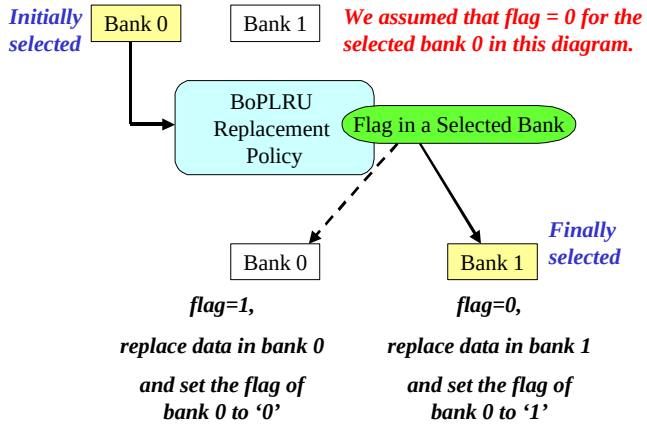
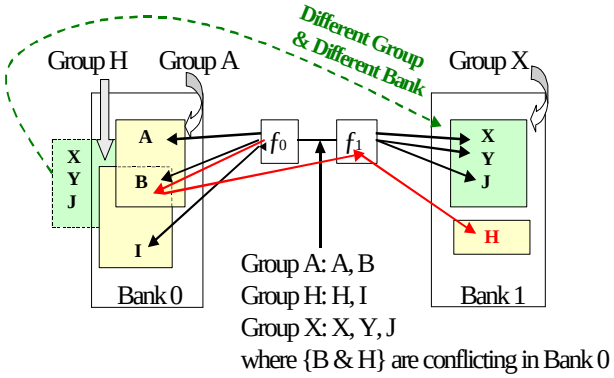



**Figure 6**. Final bank selection of BoPLRU replacement policy for a 2-way TAC scheme.

### 3.4 Placement of instructions in a TAC Scheme




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

**Figure 7.** Placement of instructions in a 2-way TAC scheme.

Figure 7 shows how the instructions from Figure 4 are written into each bank (2-way) on a cache miss. We assume that BSL selects bank 0 for Group A and H, and bank 1 for Group X initially:

- Instruction A and B of Group A are written into bank 0. We assume that the flags for each cache line for Group A are set to '0' (initial condition).
- 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 is set to '1'.
- Instruction I of Group H is written into bank 0. We assume that the flag is set to '0' (initial condition).
- The instructions X, Y, and J of Group X are written into bank 1. We assume that the flags of each cache line for Group X are set to '0'.

## 4. Experimental Environment

Figure 8 shows an overview of our simulation methodology:

- First, SPEC95INT and C++ programs were compiled by using a compiler (GNU gcc 2.6.3 and 2.7.2).
- Second, the 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 and Hedouin [8] 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, outputs such as cache miss rates, the number of instructions and data references, simulation time, etc. were collected.

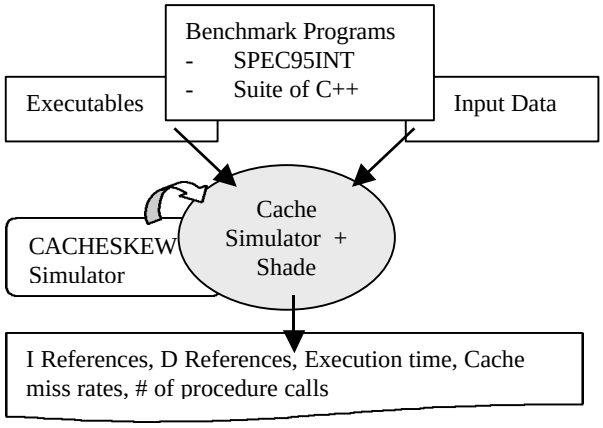



**Figure 8**. Experimental Methodology

In Figure 8, *Shade* is a tool that dynamically executes and traces SPARC v9 executables. Using Shade, the desired trace information can be specified. This means that 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, TACSim.

### 4.1 Benchmarks

Table 1 provides a description of the run-time characteristics of the benchmarks. Five of the SPEC95 integer programs were used for our simulation – gcc, go, m88ksim, compress, 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. 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 two times larger (as a harmonic mean) than that of the C++ programs.

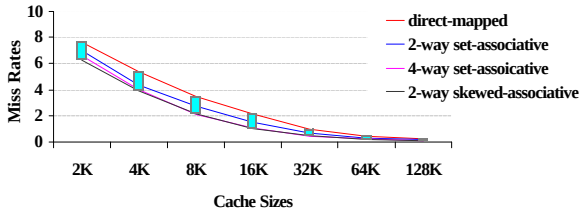
| Benchmark Program | Input | Dynamic instructions | # of Procedure calls | Instructions /call |
|---|---|---|---|---|
| SPEC95 CINT: C Programs | | | | |
| go | 2stone9.in | 584,163K | 1,610K | 362.65 |
| gcc | amptjp.i | 250,494K | 5,203K | 48.13 |
| m88ksim | ctl.raw | 850K | 16K | 50.66 |
| compress | test.in | 41,765K | 1,355K | 30.81 |
| perl | scrabble.pl scrabble.in | 63,028K | 2,611K | 24.14 |
| li | train.lsp | 189,184K | 7,971K | 23.73 |
| Suite of C++ Programs | | | | |
| deltablue | 3000 | 42,148K | 1,478K | 28.52 |
| ixx | object.h som_plus_fresco.idl | 31,829K | 1,404K | 22.65 |
| eqn | Eqn.input.all | 58,401K | 1,999K | 29.21 |
| C Harmonic mean | | 4,894K | 97K | 37.67 |
| C++ Harmonic mean | | 41,513K | 1,588K | 26.45 |

**Table 1**.Benchmark Programs Characteristics

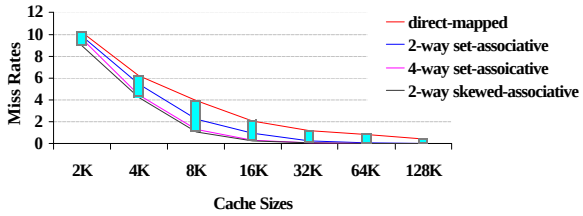
## 5. Experimental Results

The performance metrics used for comparison of different cache schemes are the instruction cache miss rates and branch prediction rates.

### 5.1 Cache Misses vs. Cache Sizes




(a) Miss Rates vs. Cache sizes for C programs (16 bytes of line size)




(b) Miss Rates vs. Cache sizes for C++ programs (16 bytes of line size)

**Figure 9.** Comparisons for cache misses according to the cache sizes.

Much previous research has been done to determine the relationship between the cache size and cache miss rates. To examine the relationship for ourselves, we simulated 4 cache schemes with C and C++ benchmark programs in Figure 9: 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 with a 16byte cache line size. The bars in Figure 9 represent the

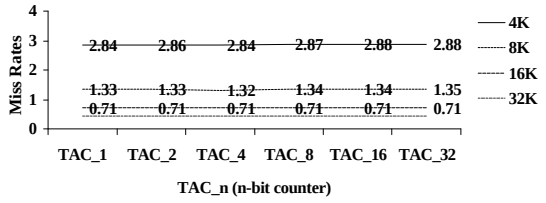
difference between the highest and the lowest miss rates for each cache size.

In Figure 9, results for the C programs show that if cache sizes are 4 Kbytes to 16 Kbytes, a more efficient cache scheme is needed since cache miss rates are significant. In the case of the C++ programs, if cache sizes are 4 Kbytes to 32 Kbytes, again a more efficient scheme for reducing cache misses is needed.

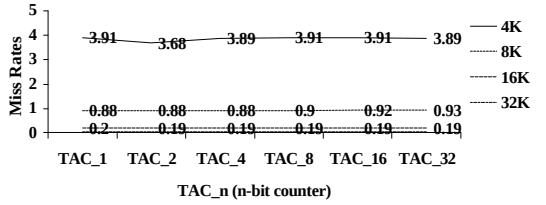
In general, if cache sizes are less than 2 Kbytes or larger than 32 Kbytes, the cache misses are similar whatever cache schemes are used. From the Figure 8, we can conclude that *it is quite reasonable to develop a more sophisticated cache scheme for reducing cache misses between 4 Kbytes and 32 Kbytes of cache sizes.*

## 5.2 Bank Switching vs. Procedure Calls

In this section, we determine the most efficient size of x-bit counter, which BSL employs for selecting banks. As we discussed in section 5.1, we primarily investigate for cache sizes which are less than 32 Kbytes. We simulated 2-Way TAC schemes with 7 benchmark programs to determine the most effective x-bit counter.




(a) Miss Rates vs. TAC_n for C programs (16 bytes of line size)




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

**Figure 10.** Cache miss rates according to the sizes of the n-bit counter.

In Figure 10, TAC_k means that BSL selects a bank for every k occurrences of 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 3, the n most significant bits of a x-bit counter represents a bank for the current instruction. Therefore, if k=2 and n=1, then we need a 2-bit counter because {00, 01} → bank 0 and {10, 11) → bank 1.

In Figure 10, we used four benchmark programs (gcc, m88ksim, li, and compress) and three C++ programs (deltablue, ixx, and eqn) to determine the most effective counter size. The results show that TAC_2 works slightly better than the others. Therefore, we recommend a small-size counter, less than 2-bit for the case of k=2 and n=1, for the BSL of a TAC scheme if a cache size is less than 32 Kbytes.

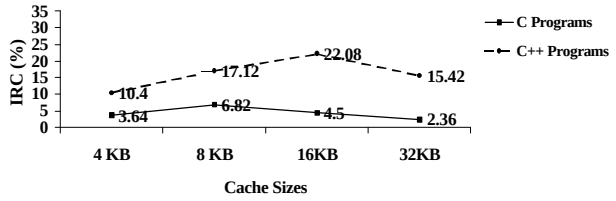
## 5.3 Skewed-associative caches vs. TAC schemes

In this section, we compare 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. The BSL was implemented with a 2-bit counter.

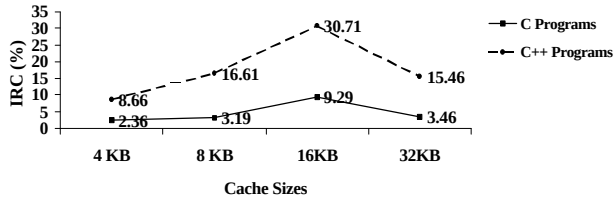
In order to compare cache miss rates between the TAC and skewed-associative caches, we used a formula called IRC, Improvement Ratio for Cache, such that:

*Cache miss rates of a 2-way skewed-associative = a;*
*Cache miss rates of a TAC scheme = b;*
*a/b = 1 + n/100* **à** *'b' has n% less cache miss rates than 'a'.*
*If n = IRC, IRC = ((a – b) / b) * 100 % ---------- (1)*

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




(a) Improvements Ratios for Cache (16 bytes of line size)

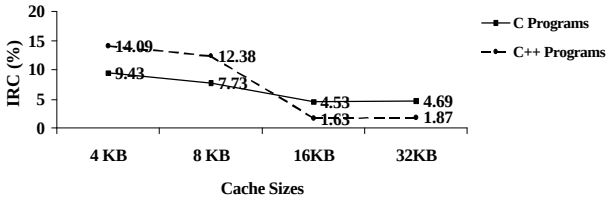



(b) Improvements Ratios for Cache (32 bytes of line size)

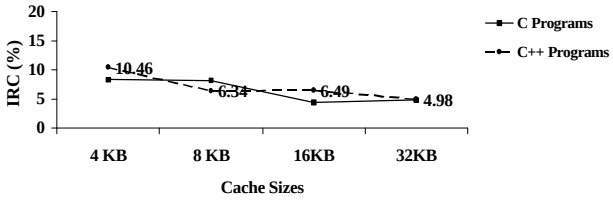
**Figure 11.** Comparisons for IRC between 2-way skewed associative and 2-way TAC.

Therefore, if we use IRC for comparing two cache schemes, we can easily get the improved result with regard to cache miss rates. In Figure 11 and 12, 4 C programs (gcc, m88ksim, li, and compress) and 3 C++ programs were used for determining IRC between skewed-associative and TAC schemes.

The results of Figure 11 show that: If the cache size is 8 Kbytes or 16 Kbytes (for C programs) or 16 Kbytes (for C++ programs), the 2-way TAC schemes can reduce cache misses much better than 2-way skewed-associative caches for cache line sizes of 16 and 32 bytes; If cache sizes are bigger than 32 Kbytes or less than 4 Kbytes, there is only a slight difference between 2-way TAC and 2-way skewed-associative schemes.




(a) Improvements Ratios for Cache (16 bytes of line size)



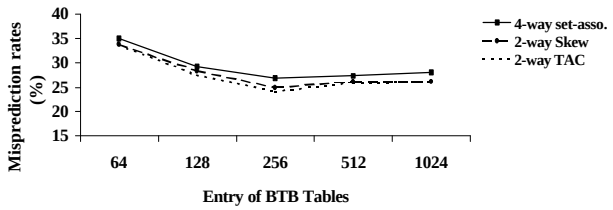

(b) Improvements Ratios for Cache (32 bytes of line size)

**Figure 12.** Comparisons for IRC between 4-way skewed associative and 4-way TAC.

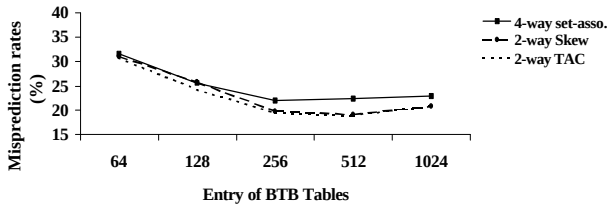
The results of Figure 12 show that: If the cache size is 4 Kbytes (for C and C++ programs), the 4-way TAC schemes reduces cache misses much better than 4-way skewed-associative caches for cache line sizes of 16 and 32 bytes; If cache sizes are larger than 16 Kbytes, the difference between 4-way TAC and 4-way skewed-associative schemes is reduced significantly since most conflict misses disappear for both the 4-way TAC and 4-way skewed-associative caches in C++ (16 bytes of line size) and C (32 bytes of line size).

## 5.4 Three 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-size BTB table that has less than 512 entries.




(a) Misprediction rates vs. Entries of BTB Tables for C Programs.




(b) Misprediction rates vs. Entries of BTB Tables for C++ Programs.

**Figure 13.** Comparisons for Misprediction rates among three cache schemes.

In this section, we determined the most effective cache scheme for BTB. We simulated the BTB with three cache schemes by using C and C++ benchmark programs in Figure 13. They are a 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.

The results of Figure 13 show that:

- For C programs, as shown in Figure 13(a), the 2-way TAC scheme for the 256-entry table of BTB works better than other sizes of the BTB tables.
- For C++ programs, as shown in Figure 13(b), the 2-way TAC scheme for the 512-entry table of BTB works better than other sizes of the BTB tables.

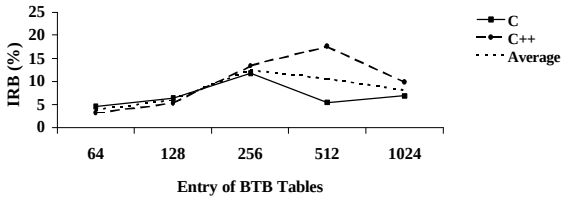
In order to compare branch misprediction rates between 2-way TAC and 4-way set-associative caches, and between 2-way TAC and 2-way skewed-associative caches, we used a formula called IRB, Improvement Ratio for Branch which is similar to IRC, such that:

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

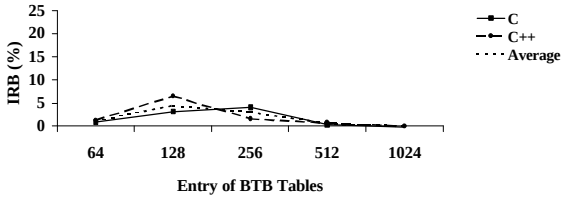
*Branch Misprediction Rates of a 2-way TAC scheme = b;*

*Then, IRB = ((a – b) / b) * 100 %* ---------- *(2)*

For Figure 14, the 2-way TAC schemes works better than 4-way set-associative caches for all table entries, from 64 entries to 1024 entries, for both C and C++ programs.




(a) 2-way TAC scheme vs. 4-way set-associative cache.




(b) 2-way TAC scheme vs. 2-way skewed-associative cache.

**Figure 14.** Comparisons for Improvement Ratios for Branch for cache schemes.

The results in Figure 14 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, 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 is a good solution for reducing branch mispredictions.

## 6. Conclusions

This paper presents a new cache scheme called TAC (Thrashing-Avoidance Cache), which can effectively reduce instruction cache misses caused by frequent procedural call/returns.

In the conventional cache schemes described in section 2, the 2-way 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 the handling of conflict misses in object-oriented programs due to the problem of frequent access to a 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 for group instructions separated by a call instruction.

Our simulation results show that:

- TAC schemes (on L1 cache) improve instruction cache miss rates by up to 9.29% for C programs and 30.71% for C++ programs over skewed-associative caches.
- 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.

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

## References

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

[Calder et al '94] 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.

[Chu '00] Yul Chu, Cache and Branch Prediction Improvements for Advanced Computer Architectures, Ph.D. Dissertation, Electrical and Computer Engineering Department, University of British Columbia, 2001.

[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 ICS, Vienna (Austria), pp76-83, July 1997.

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

[Holzle & 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.

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

[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.

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

[Seznec & Hedouin '97] Andre Seznec and Jerome Hedouin, The CACHESKEW simulator, http://www.irisa.fr/caps/PROJECTS/Architecture/CACHESKEW.html, September 1997.