2000

The 2-way Thrashing-Avoidance Cache (TAC): An Efficient Instruction Cache Scheme for Object-Oriented Languages

Yul Chu
The University of Texas Rio Grande Valley

M. R. Ito

Recommended Citation

Y. Chu and M. R. Ito, "The 2-way thrashing-avoidance cache (TAC): an efficient instruction cache scheme for object-oriented languages," Proceedings 2000 International Conference on Computer Design, Austin, TX, USA, 2000, pp. 93-98, doi: 10.1109/ICCD.2000.878273.

The 2-way Thrashing-Avoidance Cache (TAC): 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

This paper presents a new instruction cache scheme: the TAC (Thrashing-Avoidance Cache). A 2-way TAC scheme employs 2-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 policies (BoPLRU). After the BSL initially selects a bank 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. We have developed a simulation program, TACSim, by using Shade and Spixtools, provided by SUN Microsystems, on an ultra SPARC/10 processor. Our experimental results show that 2-way TAC schemes reduce conflict misses more effectively than 2-way skewed-associative caches in both C (17% improvement) and C++ (30% improvement) programs on L1 caches.

1. Introduction

During the last two decades, the memory gap problem has been steadily increasing; the speed of processors has been improving by 60% per year, whereas DRAM access time has been improving by barely 10%. The memory gap makes itself felt when a cache miss occurs and the missing word must be supplied from memory [9].

Several researchers have shown that many instruction cache misses are caused by frequent procedure call/returns in object-oriented languages. Calder et al. [2] 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 (C) programs. 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. [2], programs executing a small number of instructions in each function, such as C++, may suffer from instruction cache conflicts.

Holzle and Ungar [5] 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 it is for traditional programs. Meanwhile, the data cache misses for both programs were seen to be similar [2][5]. As a result, 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, conflict misses for a direct-mapped cache comprise about 60% of the total cache misses of a small-sized cache of 8KB [3]. 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. In this paper, we present a new cache scheme called the TAC (Thrashing-Avoidance Cache), which can effectively reduce instruction cache misses caused by call/returns.

This paper is organized as follows: Section 2 explains conflict misses; section 3 presents a new cache scheme called the TAC (Thrashing-Avoidance Cache); section 4 describes experiment environments; section 5 presents our experimental results; and section 6 provides our conclusions.

2. Conflict Misses

2.1 Conflict Miss Ratios

The miss ratios for several cache schemes are shown in Figure 1 [3]. They are obtained by using the SPEC95

benchmark suite and by implementing a cache memory (8 kilobytes capacity and 32 bytes per line).

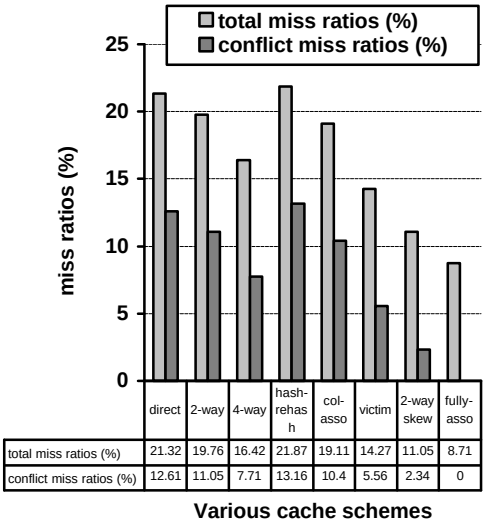

Figure 1. Miss ratios (%) of the various cache schemes.

For comparison in Figure 1, 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' indicates the total miss ratio (compulsory + capacity + conflict) while '12.61' is the computed conflict miss ratio (the total miss ratio for a scheme minus the total miss ratio for the fully-associative scheme).

From the results in Figure 1, 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 [3].

2.2 Skewing (XOR mapping) functions

Skewed associative caches have been previously proposed [7]. A 2-way skewed-associative cache consists of 2 distinct banks that are accessed simultaneously with different mapping functions. In Figure 2, 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.

Bodin and Seznec [1] presented skewing (XOR mapping) 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, and 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
- or on cache line $\sigma^2(A_1) \oplus A_2$ in cache bank 2 (on a 4-way)
- or on cache line $\sigma^3(A_1) \oplus A_2$ in cache bank 3 (on a 4-way)

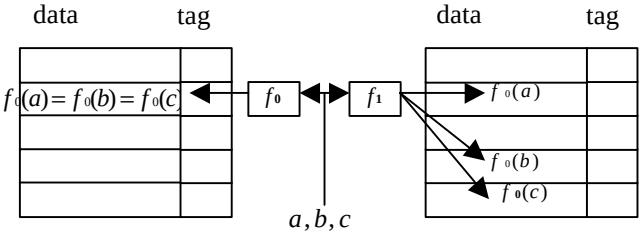

Figure 2. a, b, and c compete for the same location in bank 0, but can be present at the same time, as they do not map to the same location in bank 1 [7].

3. 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 32Kbytes) 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 by focusing 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 a TAC Scheme

A N-way TAC scheme is built with N distinct banks. In this paper, we focused only on a 2-way TAC scheme.

Since Gonzalez et al. [3] showed that XOR mapping functions work well for reducing conflict misses, TAC employs XOR mapping functions for accessing the cache memory.

In Figure 3, *a* (address) indexes into the instruction cache memory on a physical line $f_0(a)$ in bank 0 and on a $f_1(a)$ in bank 1, etc. The word data is sent to the processor if it exists in some bank. This is called a cache

hit. Otherwise, on a miss, data must be fetched from a lower level memory according to the Bank Selection Logic (BSL, refer to section 3.2) and Bank-originated Pseudo LRU replacement policy (BoPLRU, refer to section 3.3).

Each cache line consists of tag, data, and flag. The tag word consists of an address tag and some other status tags. For convenience, we represent a cache line of a TAC scheme as just data (A, B, etc) and flag (0 or 1) through this paper and omit the tag part.

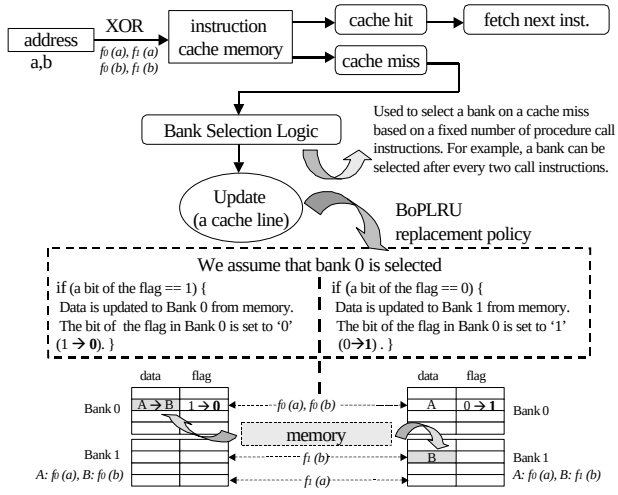

Figure 3. The basic operation of a 2-way TAC scheme.

3.2 Bank Selection Logic (BSL)

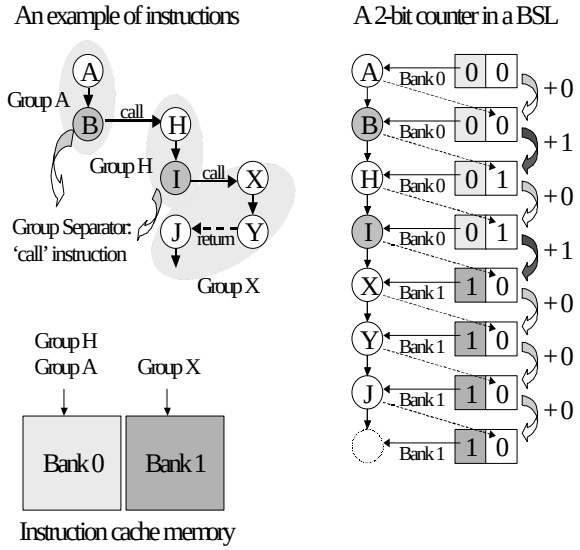

Figure 4.The operation of the BSL according to a flow of instructions (2-bit counter). Conflicts in (B, H) and (I, X).

The function of the Bank Selection Logic (BSL) is to select a bank initially on a cache miss according to a fixed frequency of procedure call instructions. The BSL employs a n-bit counter for counting the frequency of call instructions. The n-bit counter will be increased by one whenever a fetched instruction proves to be a call instruction. The first bit of the n-bit counter represents a selected bank for each instruction.

Each bank can be selected on every 2^{n-1} procedure calls. For example, if n = 2, then a bank is switched to the other bank following every two procedure calls (2^{n-1} = 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.

Figure 4 shows how a 2-bit counter in the BSL works with the flow of example instructions, which is located in the left side of Figure 4. 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 4, we assume that there are cache conflicts in (B, H) and (I, X).

3.3 Bank-originated Pseudo LRU Replacement Policy (BoPLRU)

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

The BoPLRU operation is shown in Figure 3: The BSL initially selects the bank 0 on a cache miss for b (address). Therefore, the flag of the selected line, $f_0(b)$, is read. If the flag is 1, it is set to 0 and the data fetched from memory is written into bank 0 by replacing the data in $f_0(a)$ with new data from $f_0(b)$. Otherwise, the flag is set to '1' and the missing line is written into bank 1 according to the selected line, $f_1(b)$.

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

3.4 Benefit of a TAC Scheme

Figure 5 shows how the instructions in Figure 4 are written into each bank on a cache miss. We assume that BSL selects bank 0 for Group A and H, and bank 1 for Group X:

- 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 should be 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'.

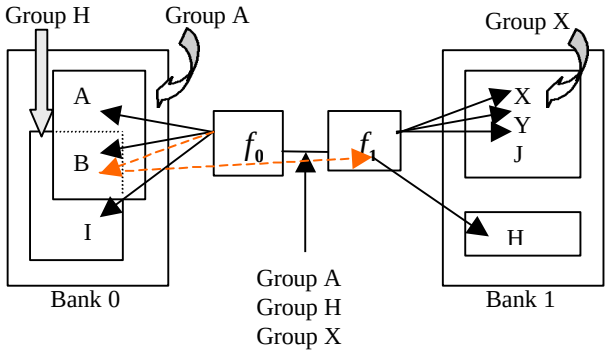

Figure 5.Placement of instructions in a 2-way TAC scheme.

If we consider the instructions in Figure 4, we can easily verify 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 quite reasonable to assume that conflict misses between instructions from different groups can occur frequently 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 [4] 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 the calling routine and the subroutine can exist in very small areas of memory.
2) Locality in Time (Temporal Locality): Handy [4] also notes that the same instructions 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 previously than one which it accessed 10,000 cycles previously.

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 to handle the calling routine and subroutine sequence 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 is increased, we expect that the TAC scheme will work even better than other conventional cache schemes.

4. Experiment Environments

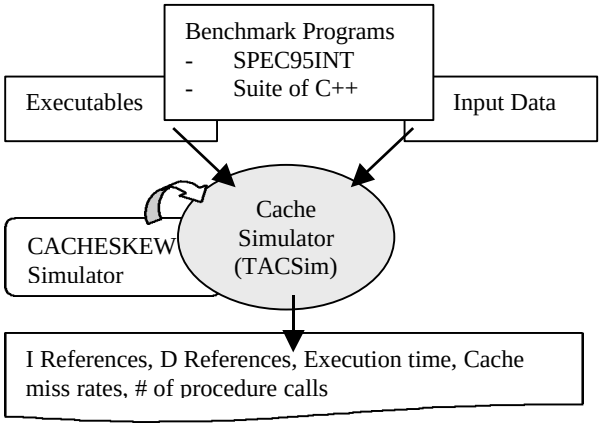

Figure 6.Experimental Methodology

Figure 6 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, output such as cache miss rates, the number of instructions and data references, simulation time, etc. were collected.

In Figure 6, *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. We can collect any detailed information for every instruction and opcode dynamically. For example, we can obtain the data for the total number of call instructions, program counter, opcode fields, etc. This information is used for our simulation tool, TACSim.

4.1 Benchmarks

Table 1 describes the benchmark programs. 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 and John [6]. The next suite of programs is written in C++ and has been used for investigating the behavior between C and C++ [2][5]. These programs are deltablue, ixx, and richards.

The table also 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 two times larger than that of the C++ programs (as a harmonic mean).

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
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
richards	1	6,778K	323K	20.97
C Harmonic mean		4,131K	81K	42.42
C++ Harmonic mean		20,264K	669K	23.64

Table 1.Benchmark Programs Characteristics

5. Experimental Results

BSL was implemented with a 2-bit counter. If the counter size of the BSL is greater than 5 bits, the instruction cache miss rates are found to be slightly higher than for a small-sized counter (less than 4 bits).

5.1 Instruction Cache Misses for Various Cache Schemes

Table 2 shows that typical programs in the C++ suite (deltablue and ixx) incur higher instruction cache misses than typical C programs (compress and li). It 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.

Bench-marks	Dir-ect	2-way	4-way	2-way skew	2-way TAC	16-way	*IR* of TAC over Skew
SPEC95 CINT (C Programs)							
go	6.869	5.503	4.871	5.453	5.378	4.920	1.37
gcc	5.934	5.115	4.172	4.123	3.964	3.452	3.86
m88-ksim	3.818	2.822	1.540	1.399	1.320	0.947	5.67
com-press	0.056	0.047	0.021	0.019	0.016	0.006	12.63
li	0.539	0.423	0.083	0.023	0.010	0.005	55.46
C++ Programs							
delta-blue	3.131	1.959	1.368	1.017	0.555	0.262	45.33
ixx	4.767	2.542	1.382	1.147	0.941	0.288	17.94
*rich-ards	12.17	4.525	4.693	2.831	2.423	2.336	14.42
C Harmonic mean							3.97
C++ Harmonic mean							20.39

* Cache size: 1 KB, line size: 16 bytes, **IR**: Improvement Rate

Table 2. Instruction cache miss rate (%) of experiment results (Cache size: 8 Kbytes, line size: 16 bytes)

Figure 7 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 C and C++ programs. The 16-way set-associative cache can be considered a good approximation to a fully-associative cache.

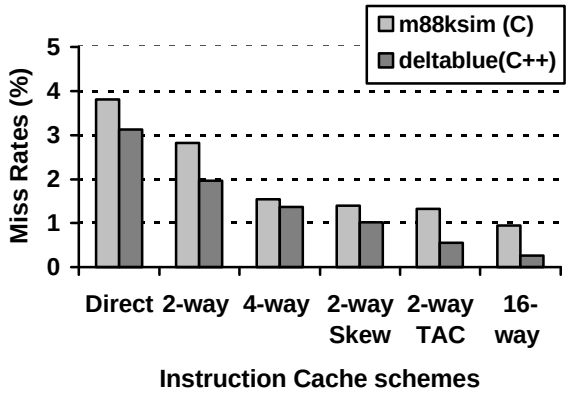

Figure 7.Comparison for instruction cache miss rates between C (m88ksim) and C++ (deltablue) programs (8Kbytes, 16bytes).

Thus, the 2-way TAC scheme can replace conventional cache schemes for traditional programs with little or no increase in hardware complexity, and is even more suited to object-oriented programs than conventional caches.

Table 3 also shows that the TAC scheme removes conflict misses more effectively than the 2-way skewed-associative cache scheme in ***both C (17% improvement)***

and C++ (30% improvement) programs. For the results of Table 3, we simulated the benchmark programs with 16 Kbytes of the cache size, and 16 and 32 bytes of cache line size.

Bench-marks	Dir-ect	2-way	4-way	2-way skew	2-way TAC	16-way	**IR** of TAC over Skew
SPEC95 CINT (C Programs)							
go	4.031	3.689	2.265	2.233	2.292	1.843	-2.64
gcc	3.990	2.793	2.342	2.236	2.157	1.986	3.53
m88-ksim	2.434	1.167	0.846	0.703	0.673	0.472	4.32
com-press	0.039	0.017	0.013	0.008	0.008	0.006	2.29
li	0.298	0.061	0.003	0.004	0.003	0.002	20
C++ Programs							
delta-blue	1.649	0.662	0.299	0.239	0.167	0.025	30.37
ixx	2.512	1.292	0.157	0.138	0.037	0.011	72.82
*rich-ards	12.17	4.525	4.693	2.831	2.423	2.336	14.42
C Harmonic mean							8.03
C++ Harmonic mean							25.86

* Cache size: 1 KB, line size: 16 bytes, **IR**: Improvement Rate

(a) Cache size: 16 Kbytes, line size: 16bytes

Bench-marks	Dir-ect	2-way	4-way	2-way skew	2-way TAC	16-way	**IR** of TAC over Skew
SPEC95 CINT (C Programs)							
go	2.323	2.136	1.328	1.311	1.391	1.080	-6.10
gcc	2.825	1.989	1.663	1.672	1.602	1.369	4.18
m88-ksim	1.802	0.877	0.677	0.490	0.457	0.324	6.70
com-press	0.028	0.012	0.010	0.007	0.005	0.004	22.53
li	0.199	0.050	0.019	0.005	0.002	0.001	56.14
C++ Programs							
delta-blue	1.202	0.676	0.273	0.185	0.046	0.015	75.14
ixx	1.882	1.050	0.139	0.214	0.079	0.010	62.77
*rich-ards	12.17	4.525	4.693	2.831	2.423	2.336	14.42
C Harmonic mean							17.45
C++ Harmonic mean							30.43

* Cache size: 1 KB, line size: 16 bytes, **IR**: Improvement Rate

(b) Cache size: 16 Kbytes, line size: 32bytes

Table 3. Instruction cache miss rate (%) of experiment results.

6. Conclusions

We have discussed several cache schemes for reducing conflict misses: direct-mapped, 2-way set-associative, 4-way set-associative, hash-rehash, victim, and 2-way skewed-associative. The 2-way skewed-associative cache offers the lowest miss ratio among previous schemes, and is significantly lower than that of a 4-way set-associative cache. However, the 2-way skewed-associative cache has a limitation in handling the conflict misses due to the large number of small functions in object-oriented programs. The main reason for this is that the 2-way skewed-associative cache is designed for reducing the conflict misses of the individual instructions only and not groups of instructions. However, the TAC scheme works for not only the individual instructions but also the group of instructions (calling routine and subroutine). Our simulation results show that:

- The TAC scheme reduces the instruction cache miss rates more than 17% (SPECint95 programs, C programs) and 30% (a suite of C++ programs) over the 2-way skewed-associative cache.
- The hardware cost and memory access time are similar to the 2-way set-associative cache since the TAC scheme employs 2-way banks and the simple hardware complexity of XOR mapping functions.
- The TAC scheme uses the XOR mapping functions to manage all cache memory locations efficiently.

References

[1] F. Bodin, A. Seznec, Skewed-associativity enhances performance predictability, Proc. of the 22th *Int. Symp. on Computer Architecture*, Santa-Margharita, June 1995

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

[3] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa, Eliminating cache conflict misses through XOR-based placement functions, Proc. of *Int. Conf. on Supercomputing*, Vienna (Austria), July 1997, pp76-83.

[4] Jim Handy, The Cache Memory Book, Academic Press, Inc., 1993.

[5] 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.

[6] 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.

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

[8] A. Seznec and J. Hedouin, The CACHESKEW simulator, http://www.irisa.fr/caps/PROJECTS/Architecture/CACHESKEW.html,September 1997.

[9] Maurice Wilkes, The Memory Gap, Proceedings of Workshop on Solving the Memory Wall Problem, Held in Conjunction with ISCA-2000, June 11, 2000, keynote.