Appeared in the 3rd Workshop on Interaction between Compilers and Computer Architectures, October 1998.

A Comparison of Software Code Reordering and Victim Buffers

Iris Bahar[†] Brad Calder[‡] Dirk Grunwald[*]

[†]Division of Engineering, Brown University
[‡]Department of Computer Science and Engineering, University of California, San Diego
[*]Department of Computer Science, Univeristy of Colorado, Boulder

Abstract

Instruction cache performance is critical to instruction fetch efficiency and overall processor performance. The layout of an executable has a substantial effect on the cache miss rate during execution. This means that the performance of an executable can be improved significantly by applying a code-placement algorithm that minimizes instruction cache conflicts. Alternatively, the hardware configuration of the instruction cache itself may greatly influence cache performance. For instance, increasing associativity or selective placement of data in the cache may significantly reduce conflict misses as well.

In this paper we compare the performance benefits of compiler-based code-placement algorithms to hardware-based schemes using a full simulation model of an out-of-order, 4-way issue processor. We compare the effectiveness of code-placement algorithms when applied to self-trained and cross referenced profiling data. We compare the benefits of adding small fully associative buffers, including variants of victim buffers, to be used along side the first level instruction cache. Our results show that code placement achieved an average 10% speedup, victim buffers obtained a 15% speedup on average, and combining both of these resulted in a net 20% speedup on average.

1 Introduction

The performance of today's high-performance microprocessors continues to grow. This increase in performance to due in part to the use of speculative, out-of-order execution coupled with highly accurate branch prediction. Branch prediction has increased instruction-level parallelism (ILP) by allowing programs to speculatively execute beyond control boundaries, while out-of-order execution has increased ILP by allowing more flexibility in instruction issue and execution. The combination of these techniques has increased processor performance in part by hiding the memory latency penalty in the case of a cache miss.

Out-of-order execution masks some of the latency of data cache misses by executing other instructions while the memory reference is resolved. A large central instruction window allows sufficient instructions to be "in-flight" such that an instruction cache miss can be tolerated.

Despite these advantages, there are many impediments to efficient instruction fetch and dispatch on modern processors. High-speed instruction caches are typically direct-mapped, and suffer from cache conflicts; this typically occurs for procedure calls, and is exacerbated by current implementations of modern programming languages [4]. Futhermore, aliasing conflicts occur in branch predictors, and branch target buffers are of a limited size and suffer from capacity misses. Speculative execution complicates the analysis of instruction fetch because speculative instructions must be fetched into the instruction cache, even though those instructions may never contribute to the final program execution.

Since instruction execution is felt to be predictable [17, 5], several researchers have proposed software methods to reorganize applications. Alternatively, the performance of direct mapped caches can be improved by the addition of small full-associative buffers, such as the victim cache proposed by Jouppi [10]. To date, no study has compared the effectiveness of these software and hardware mechanisms. The software mechanisms have the advantage of being able to *avoid* cache conflicts at the expense of profiling, code modification and the fallibility of selecting a profiling training set. Software mechanisms can affect both the instruction cache and the branch architecture – for example, "aligning" branches can improve the performance of the branch target buffer [3] and improve branch predictor performance [18]. Furthermore, properly aligning instructions on cache line boundaries improves instruction fetch efficiency, because more of the fetched cache line is used by the processor. The hardware mechanisms have the advantage of reducing the cost of cache misses with no need for profiling, but this requires more hardware resources and cannot (directly) improve the branch and fetch performance.

In this study, we use a cycle-level simulator of a four-issue, speculative, out-of-order processor to assess the effectiveness of these software code placement and associative-buffer solutions. We make the following contributions:

- We measure the effectiveness of code layout and associative-buffer solutions using a cycle-level simulator, and report the performance improvements as improvements in instructions-per-cycle. We show that code-layout can produce speedups of up to 60% for our processor configuration.
- We show that variants of the victim buffer design are

more effective than those proposed by Jouppi *et al.* for the instruction caches we considered. These designs are also simpler to implement.

We describe the related work on code layout algorithms and victim buffers in §2. Section 3 describes the simulation environment and benchmarks we used. In §4, we use the simulation model to select between different implementations of "associative buffers," selecting an organization called the "penalty buffer" for use in the remainder of the study. We then compare the effectiveness of the code layout algorithm, explain why it encounters problems and then see how to combine the associative buffers with the code layout to mitigate those problems.

2 Background and prior work

We first review the organization of a victim buffer as originally proposed by Jouppi. We then expand on the procedure placement algorithm we used.

2.1 Associative Buffers for the Instruction Cache

The *victim cache* was originally presented by Jouppi in [10]. We describe modifications to this basic design in §4.1. On a main cache miss, the victim cache is accessed; if the address hits in the victim cache, the data is returned to the CPU and, at the same time, it is promoted to the main cache. The replaced line in the main cache is moved to the victim cache, therefore performing a "swap". If the victim cache also misses, an L2 access is performed; the incoming data fills the main cache, and the replaced line moves to the victim cache. The replaced entry in the victim cache is discarded and, if dirty, written back to the second level cache. Jouppi showed that adding a small victim buffer could gain much of the performance advantage of a 2-way set-associative cache. His analysis, however, was not done on a full simulation model and performance improvement was measured only in terms of reduced miss rate. Moreover, the impact on performing of doing the swap was not taken into account

In Jouppi's scheme, the victim cache was filled only after a line of data was evicted from the main L1 cache. Buffers need not be used only for holding "evicted" data. Instead, specific fills from the L2 to the L1 cache may be sorted out and placed in the buffer directly rather than being written first to the L1 cache. This variation of the victim buffer scheme may help reduce the need for swapping between the buffer and the L1 cache. This and other modifications to the basic victim cache design will be discussed in more detail in §4.1.

2.2 Procedure Placement Algorithms

Many software techniques have been developed for improving instruction cache performance. Techniques such as basic block re-ordering [9, 13], function grouping [7, 8, 9, 13], reordering based on control structure [12], and reordering of system code [16] have all been shown to significantly improve instruction cache performance. The increasing latency of second-level caches means that cache conflicts have a dramatic effect on program performance.

Recent work on procedure placement to improve instruction cache performance shows that further improvements in performance are achieved by keeping track of where cache lines procedures are placed to eliminate conflict misses, and by using temporal information to guide the placement algorithm [8, 7, 11]. This research showed that the cache miss rate is significantly reduced by taking the cache attributes into consideration when performing the placement and using the temporal relationships between procedures. In this paper we used the temporal procedure placement approach of Gloy *et al.* [7]. The temporal procedure placement algorithm uses a temporal relationshiop graph profile [7], the size of the procedures, and the structure of the target cache (i.e., cache size and block size) to eliminate cache conflicts and increase cache line utilization. See [7] for a complete description of the placement algorithm.

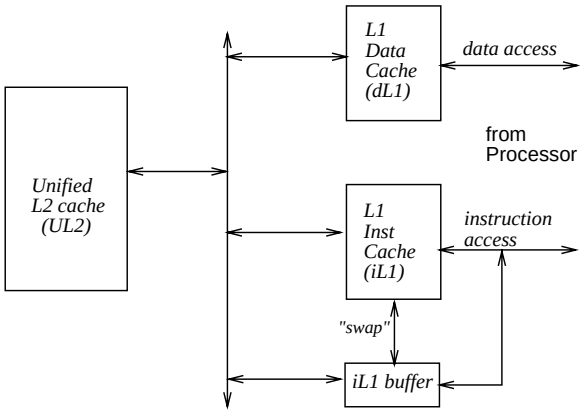

Figure 1: Configuration of the memory hierarchy in the base simulator

3 Experimental setup

We use an extension of the *SimpleScalar* [2] tool suite to perform the experiments described in this paper. SimpleScalar is an execution-driven simulator that originally used binaries compiled to a MIPS-like target; recently, a version of SimpleScalar has been modified to use the Digital Alpha instruction set. SimpleScalar can accurately model a high-performance, dynamically-scheduled, multi-issue processor. We use an extended version of the simulator that accurately models the memory hierarchy. The model implements non-blocking caches and complete bus bandwidth and contention modeling [1]. Other modifications were added to handle precise modeling of cache fills, various victim buffer organizations and code layout transformations.

We used this version of SimpleScalar because it allows us to use the ATOM [15] instrumentation and analysis system. We used ATOM to determine the code layout transformations, and modified SimpleScalar to implement the code layout.

Figure 1 shows the organization of the victim cache in the memory hierarchy. From the figure, note that an instruction access from the CPU will go directly to the I-cache (iL1), and optionally may also go to the iL1 buffer. Likewise, both the iL1 buffer and the main iL1 cache have access to the UL2 data bus. To implement Jouppi's victim cache, an instruction accesses is first made to the iL1. The victim cache is accessed on an iL1 miss. If the address hits in the victim cache, the instruction is returned directly from the victim cache to the CPU and then promoted to the iL1 by swapping it with replaced line in the iL1. On a victim cache miss, fill data from the UL2 is written to the iL1 (which returns the data to the CPU) and evicted iL1 data is written to the victim cache. Swapping takes place as shown in Figure 1. Note that with this hardware configuration, it is possible to simultaneously lookup data in

both the iL1 and the iL1 buffer. The advantage of this will be discussed in § 4.1.

Tables 1, 2, and 3 show the configuration of the processor modeled. Note that first level caches are on-chip, while the unified second level cache is off-chip. In addition we have a buffer associated with the first level instruction cache, as shown in Figure 1; this buffer was implemented as a fully associative cache with LRU replacement. In our experiments we varied the size of the buffer from 8 entries to 32 entries. Note that we chose an 8K first level instruction cache configuration in order to obtain a reasonable hit/miss rate from our benchmarks [6]. In Tables 2 and 3 note that some types of resource units (e.g., the FP Mult/Div/Sqrt unit) may have different latency and occupancy values depending on the type of operation being performed by the unit. Finally, in Table 4 we provide details on the IPC, miss rate, and input set for the benchmarks using the baseline configuration.

Table 1: Machine configuration parameters.

Parameter	Configuration
L1 Icache	8KB direct; 32B line; 1 cycle lat.
L1 Dcache	32KB direct; 32B line; 1 cycle lat.
L2 Unified Cache	128KB direct; 32B line; 2 cycles
Memory	64 bit-wide; 20 cycles on page hit, 40 cycles on page miss
Branch Pred.	4k gshare + 4k bimodal + 2k meta
BTB	512 entry 8-way set assoc.
Return Addr. Stack	32 entry queue
ITLB	32 entry fully assoc., 4KB mapping
DTLB	64 entry fully assoc., 4KB mapping

Table 2: Processor resources.

Parameter	Units
Fetch/Issue/Commit Width	4
Integer ALU	4
Integer Mult/Div	2
FP ALU	2
FP Mult/Div/Sqrt	1
DL1 Read Ports	2
DL1 Write Ports	1
Instruction Window Entries	64
Load/Store Queue Entries	32
Fetch Queue	16
Minimum Misprediction Latency	6

4 Analysis and Results

The following section describes in detail the various experiments run and evaluates their results. All experiments were performed assuming an 8K direct mapped instruction cache as explained in Section 3.

4.1 Selecting an Associative Buffer Organization

We made several modifications to the victim buffer described in §2. First, we assumed that we could examine the tags in the direct-mapped cache *and* the victim buffer concurrently. Since the victim buffer is typically small, we felt this was unlikely to impact the instruction fetch cycle time. This implementation still swaps the data between the victim cache and the instruction cache. Since we are only examining instruction caches, there are no dirty lines to be written back to memory. We refer to this algorithm as *victim cache swapping* (VCS), because swapping is performed on a victim cache hit. Since the instruction cache is accessed every cycle, swapping the entries in the victim cache and the instruction cache may cause instruction fetch to stall while the swap occurs. However, the swapping should result in a lower miss rate, as demonstrated by Jouppi [10].

The process of swapping instructions between the victim cache and iL1 ties up both caches for one cycle. Although swapping may improve the iL1 miss rate, it is possible that the extra time required to do the swap may hurt performance. For this reason, we tried a variation on the algorithm that does not require swapping – i.e. on a victim cache hit, the line is not promoted to the main cache. We refer to this configuration as *victim cache non-swapping* (VCN). Once a block is placed in the victim cache, the only way it can move to the I-cache is if it is replaced and filled from the UL2 to the I-cache. However, instruction fetch requests can be satisfied as long as the instructions are still in the victim cache.

Buffers need not be used only for holding "evicted" data. Instead, specific data or instructions may be sorted out and placed in the buffer rather than being written to the L1 cache. We next present a different associative buffer policy that assigns cache lines either to the I-cache or associative buffer based on instruction "criticality". We have observed that, as opposed to data cache behavior, there is not a fixed correlation between variations in instruction cache miss-rate and performance gain. This is due in part to the fact that some misses are more critical than others. The miss latency often may be hidden by the processor if it can operate on other instructions found in the fetch/dispatch queue or instruction window while the miss is being serviced. Therefore, an I-cache miss that occurs while either the fetch/dispatch queue or instruction window is relatively full, may be considered "non-critical" since the processor may have a choice of instructions to execute

Table 3: Latency and occupancy of each resource.

Resource	Latency	Occupancy
Integer ALU	1	1
Integer Mult	3	1
Integer Div	20	19
FP ALU	2	1
FP Mult	4	1
FP Div	12	12
FP Sqrt	24	24
Memory Ports	1	1

		Reference Input		
Program	Details	Inst. Per Cycle	I-Cache Miss Rate	Input Set
m88ksim	SPEC95, Peak	1.23	4.4%	ref
perl	SPEC95, Peak	1.39	3.3%	ref (primes)
gcc	SPEC95, Peak	0.74	4.7%	ref (varasm.I)
go	SPEC95, Peak	1.09	3.1%	ref
vortex	SPEC95, Peak	0.81	6.5%	ref (persons.1k)
groff110	Gnu Groff V1.10, DEC C++ V6.20	0.75	7.1%	MAN man pages

Table 4: Details of our benchmark applications using the baseline configuration.

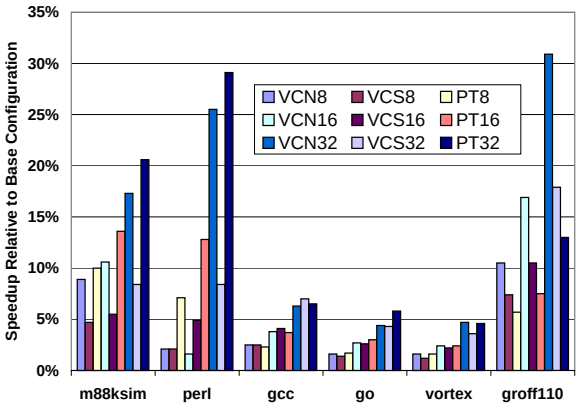

Figure 2: A comparison of three different associative buffer management policies: Victim Cache with Swapping (VCS), Victim Cache with No Swapping (VCN), and the Penalty Buffer (PT). The graph shows the relative speedup in IPC when using 8, 16 or 32 entries.

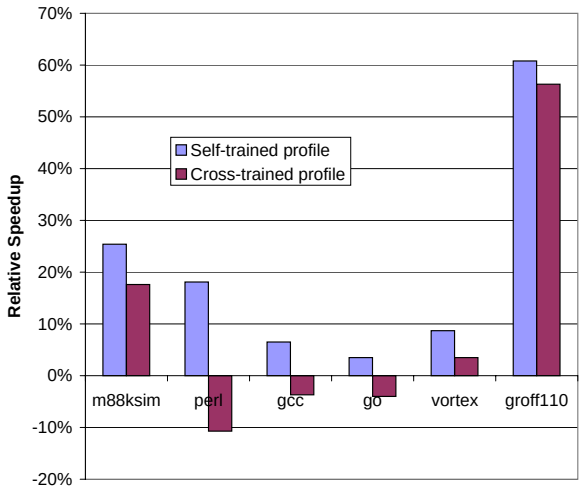

Figure 3: The relative performance improvement for code layout using two different profiles, for each benchmark. For the first case, marked "self-trained profile", the same input was used for profiling and execution. For the second case, marked "cross-trained profile", different inputs were used.

next and the miss will tend not to stall the processor. Alternatively, if a miss occurs while the queues are relatively empty few choice are available and therefore more likely to cause a stall. Moreover, since misses sometimes present a burst behavior, these queues can remain empty for several cycles and all latency is detrimental to performance.

In order to reduce the occurrence of "critical misses", we implemented a scheme using a *penalty buffer* (PT) along with the I-cache. We monitor the number of valid entries in the instruction window (*i.e.* the Resource Reservation Unit (RUU) in Simplescalar terminology). Upon an I-cache miss, we mark the misses as either *critical* (RUU with few valid entries) or *non-critical* (RUU with many valid entries) according to the status of the RUU. When a fill comes back from the UL2, we place *non-critical* instruction misses in the *penalty buffer* (*i.e.* the associative buffers) and *critical* ones in the I-cache. In this way we preserve the instructions contained in the main cache that are presumed to be more critical. Using an RUU with 64 entries, we determined through various experiments that if a miss occurred when less than 32 of the 64 entries were valid, then the miss could be considered *critical*. If the RUU had more than 32 valid entries at the time of the miss, we bypass the I-cache and place the fill in the penalty buffer.

Figure 2 shows the performance increase, measured as relative improvements in the IPC, for the different buffer management policies and buffer sizes. All comparisons are made with respect to the base case as detailed in Section 3. We experimented with three different buffer sizes (8-, 16-, and 32-entry) and three different buffer management policies for a total of nine experiments. Results marked *VCS* refers to experiments run using a victim cache with swapping while *VCN* refers to a victim cache with no swapping. Finally, *PT* refers to experiments using the *penalty buffer* scheme.

The first thing to note from the results is that a victim cache with swapping consistently gives similar or worse results when compared to a victim cache without swapping. This trend is found across all three victim cache sizes we tried. For instance *m88ksim* sees a 39% performance improvement using a 16-entry victim cache without swapping, but only a 20% improvement using the same sized victim cache with swapping. This result is attributed to the fact that any pending accesses to the I-cache must be stalled one cycle if a victim/iL1 swap is in progress. This hurts performance by preventing pending instructions from being accessed sooner. Moreover, it may prevent speculative wrong path instructions from beginning execution since branches may now be resolved before the speculative I-cache accesses can be made. While this may eliminate some needless accesses to the I-cache, it has been shown that wrong path instruction prefetching often improves performance [14].

The advantage of using the associative buffer as a *penalty buffer* instead of a victim cache (without swapping) is not so clear cut, although across all the benchmarks, the penalty buffer scheme tends to provide an advantage over the victim cache on for a buffer of size 8 or 16 entries. The reason for these mixed results may be due to using a simple measurement of "criticality" such as the number of valid RUU entries. This measurement does not give a precise model of the state of the pipeline, thus creating interference in the penalty buffer mechanism. Furthermore, an instruction that was deemed non-critical in the past, may be more critical the next time it is accessed. In such a case, we would be better of using the buffer as a victim cache.

4.2 The performance of code-layout transformations

Figure 3 shows the relative speedup for procedure placement when using the same input to both create the placement and the results (SELF), and when using different inputs (CROSS). Results show that significant performance improvements (20% for *m88ksim* and 60% for *groff*) can be achieved from SELF code placement. Performance decreases for *perl* because of marked differences between inputs. For CROSS results, performance also decreases for *gcc* and *go*. This decrease comes from 2nd level cache misses, as can be seen in Figure 4.

Figure 4 shows the 2nd level cache miss rates for the BASE, SELF and CROSS runs. In addition, it shows the miss rates if the speculative (wrong path) instructions were not fetched from the cache (No Fetch). These results show that the

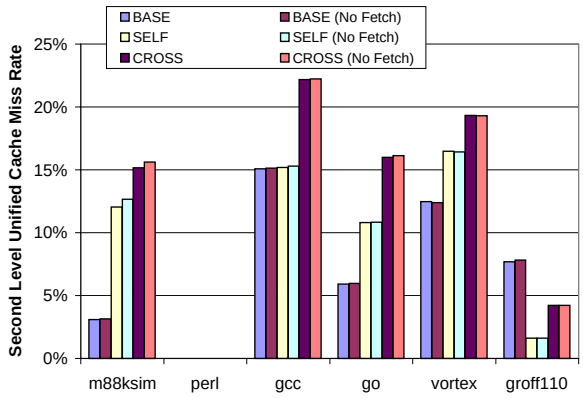

Figure 4: The cache miss rate the second level unified cache. We show the baseline configuration, the self-profiled configuration and the cross-profiled configuration. For each configuration, we show the miss rate when speculative instruction fetch is enabled and when it is disabled (No Fetch).

Program	*SELF*	*CROSS*	*2-Way*	*PT8*	*PT16*	*VCN32*
m88ksim	25.4%	17.6%	14.0%	10.0%	13.6%	17.3%
perl	18.1%	-10.7%	24.5%	7.1%	12.8%	25.5%
gcc	6.5%	-3.7%	4.6%	2.3%	3.7%	6.3%
go	3.5%	-4.0%	6.5%	1.7%	3.0%	4.4%
vortex	8.7%	3.5%	2.9%	1.6%	2.4%	4.7%
groff110	60.8%	56.3%	32.4%	5.7%	7.5%	30.9%

Table 5: Speedup relative to the base configuration using the code layout algorithm and different numbers of associative buffers.

second level misses increase over the baseline architecture for all the programs except *groff* and *perl* (note that *perl* has an effective 0% miss rate). This shows that improved placement algorithms are needed to provide combined code and data second level cache placement.

4.3 Comparing code-layout and associative buffer organizations

In Table 5 we compare the performance improvement relative to the base configuration using the code-layout algorithm and different associative buffers schemes. We show only the best buffer configurations for each size of buffer considered. For instance, from the results given in Figure 2, the best performance for an 8-entry buffer was obtained using the *penalty buffer* scheme, therefore PT8 is listed in the Table.

Overall, the self-trained code layout scheme gives the best performance improvement. This can be expected since we are optimally placing blocks of code based on flow diagrams for a particular input set. When different inputs are applied to the benchmarks, the code layout scheme gives mixed results; for some applications such as *groff110* the performance is almost as good as the self-trained profile, but for others such as *perl*, the performance is worse than the base case. This correlates with the results given in Figure 3.

The advantage of the hardware based buffer scheme is that its performance is not very sensitive to the given input set. In all cases shown in Table 5 which use a buffer we have a performance improvement compared to the base case, although not as great as with the code layout scheme. However, only for

	2 Way	*Self-Trained Layout*				*Cross-Trained Layout*			
	Assoc.	*Base*	*PT8*	*PT16*	*VCN32*	*Base*	*PT8*	*PT16*	*VCN32*
m88ksim	14.0%	25.4%	28.8%	30.6%	31.5%	17.6%	21.4%	23.1%	26.5%
perl	24.5%	18.1%	22.7%	36.0%	50.0%	-10.7%	-1.9%	7.8%	28.1%
gcc	4.6%	6.5%	7.5%	8.3%	8.5%	-3.7%	-2.8%	-2.1%	-2.2%
go	6.5%	3.5%	4.0%	4.6%	4.6%	-4.0%	-3.3%	-2.6%	-3.2%
vortex	2.9%	8.7%	9.2%	9.6%	10.5%	3.5%	3.5%	3.7%	5.1%
groff110	32.4%	60.8%	65.1%	67.4%	67.6%	56.3%	59.9%	61.9%	63.0%

Table 6: Speedup relative to the base configuration using the code layout algorithm combined with different numbers of associative buffers

the 32-entry victim cache scheme do we achieve the same performance improvements as an 8K 2-way associative I-cache. This is most likely due to the fact that a large percentage of the I-cache misses are due to conflicts. A code layout scheme has the advantage of *avoiding* the conflict (at the expense of profiling) which can improve the miss rate as well as the instruction fetch and branch predictor efficiency. The hardware scheme, on the other hand, requires no profiling, but can only indirectly improve the branch and fetch performance.

A small fully associative buffer will alleviate some of these conflict problems, but not as well as a 2-way I-cache. However, increasing the size or associativity of the I-cache is often not possible without also increasing the cycle latency which may cause an overall decrease in processor performance. Therefore, a small associative buffer may be a more realistic hardware solution.

4.4 Combining code-layout and associative buffer organizations

As shown in Table 5, performance gains using the code layout scheme may not carry over when using a different input set. In these cases, victim buffers can potentially provide significant benefits.

Table 6 shows the performance improvement when code layout and buffers schemes are combined. The columns labeled *Base* refer to performance improvement compared to the base case using code layout alone without the use of buffers. As with Table 5, we show only the best buffer configurations for each size buffer considered, since the trend stayed the same even with code-layout optimization initially applied. One interesting thing to note from these results is that for some applications the buffer can improve performance even more after code layout optimization is applied than when it is used without the code optimization. For instance, *perl* improves 18% by code layout optimization alone (using the self-trained inputs) and 25% using a 32-entry victim cache (VCN32) alone. However, if these two schemes are combined, performance improves by 50% (more than the sum of the two previous improvements). This improved use of the victim cache is also seen when combined with the cross-trained layout; without the buffers, the performance of *perl* degrades by 10.7%; combined with the victim cache, the performance improves by 28%. This small associative buffer can be thought of as a correction mechanism when code layout optimizations do not carry over well to different input sets. Moreover, the buffers themselves may become more effective at improving performance after code optimization is applied since the new layout may avoid some conflicts and enable the buffers to hold data longer (thereby increasing its chances of being accessed again).

5 Conclusions

In this paper we have compared the performance benefits of compiler-based code-placement algorithms to hardware-based schemes using a full simulation model of an out-of-order, 4-way issue processor. In particular, we compared code layout performance to hardware-based schemes that use variations of Jouppi's victim cache.

Of the hardware schemes considered, we found that implementing a victim cache *without swapping* not only offered a clear advantage in performance to a swapping scheme but also simplified the implementation. Using the victim cache as a "penalty buffer" also offered some advantages, particularly for smaller sized buffers.

We have shown that code placement algorithms offer the greatest potential in performance improvement (up to 60%). The results also show that a complete memory hierarchy code and data placement solution is needed. Concentrating on first level cache misses reduced the miss rate from 5% down to 4% on average, but at a cost of increasing the second level cache miss rate from 7.5% up to 12.6% on average.

The results further show that the improvements from code placement may not be obtainable when running the applications on significantly different inputs from the training set. In such a case, victim caches can serve as a "correction mechanism" when used in combination with code placement. Using the victim cache by itself achieved a 15% speedup on average, while the code placement achieved an average 10% speedup. When combining both of the techniques together an average 20% speedup was achieved.

References

[1] D. Burger and T. M. Austin. Simplescalar tutorial. presented at *30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December, 1997.

[2] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report TR#1308, University of Wisconsin, July 1996.

[3] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1994.

[4] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994. Also available as University of Colorado Technical Report CU-CS-698-94.

[5] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Mass., October 1992. ACM.

[6] Jeffrey Gee, Mark Hill, Dinoisions Pnevmatikatos, and Alan J. Smith. Cache performance of the spec benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[7] N. Gloy, T. Blockwell, M.D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *30th International Symposium on Microarchitecture*, December 1997.

[8] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the SIGPLANS'97 Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[9] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the16th International Symposium on Computer Architecture*, pages 242–251, June 1989.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ACM, 1990.

[11] John Kalamatianos and David Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *4thIntl. Symp. on High Performance Computer Architecture*, February 1998.

[12] S. McFarling. Procedure merging with instruction caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):71–79, June 1991.

[13] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.

[14] Jim Pierce and Trevor Mudge. Wrong-Path Instruction Preefetching. In *29th Annual Intl. Symp. on Microarchitecture*, Paris, France, December 1996. ACM, IEEE.

[15] 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*. ACM, ACM, 1994.

[16] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *1stIntl. Symp. on High Performance Computer Architecture*, pages 360–369, January 1995.

[17] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 59–70, Toronto, Ontario, Canada, June 1991.

[18] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.