Hybrid Compiler and Microarchitecture Technique for Cache Traffic Optimization

Mohamed Zahran
Department of Electrical Engineering
City College of New York of
City University of New York
New York, NY 10031
mzahran@ccny.cuny.edu

Anasua Bhowmik
Department of Computer Science
and Automation
Indian Institute of Science
Bangalore, India
anasua@csa.iisc.ernet.in

Abstract

Memory system is one of the main performance-limiting factors in contemporary processors. This is due to the gap between the memory system speed and the processor speed. This results in moving as much memory as possible from off-chip to on-chip. Furthermore, we are on a sustained effort into integrating a larger number of devices per chip. This renders integrating a large on-chip memory feasible. However, cache memories are starting to give diminishing returns. One of the main reasons for that is the delay in writing back the data of the replaced block to memory or to the next level cache. This makes block replacement time consuming, and therefore affects the overall performance. In this paper, we present a compiler-microarchitecture hybrid technique for solving the cache traffic problem. The microarchitecture part deals with bandwidth management. This is done by predicting the time at which a dirty cache block will no longer be written before replacement, and writing it back to the memory, at the time of low traffic. Thus, when the block is replaced, it is clean and the replacement is done much faster. The compiler technique deals with bandwidth saving. The compiler detects values that are dead*, and hence do not need to be written to the memory altogether. Therefore, reducing the traffic to the memory and making the replacement faster. We show that the proposed techniques reduce the writebacks from L1 cache by 24% for SpecINT and 18% for SpecFP. Moreover, around half of the dirty blocks are cleared during low traffic time, and before their actual replacement time.*

1 Introduction

Memory system is presenting the Von-Neumann bottleneck, and is limiting the performance due to the consistent increase in the gap between memory speed and processor speed. Cache memories have received great attention as a simple and efficient way of reducing this gap.

The advances in process technology have resulted in having several levels of cache memory on-chip. For example, the Intel Itanium processor has three levels of on-chip caches, for a total of 3 MB [21] and L3 cache is taking about 43% of the die area.

As Moore's law is expected to be valid for at least a decade to come, the semiconductor industry association (SIA) predicts that process technology advancement will continue until we hit a physical limit [2]. This means having either more cache levels or bigger cache size.

However, the cache memory performance starts to give diminishing return. One important reason which prohibits the increase in cache performance, is the traffic between the cache in a certain level and the cache in the next level, or the main memory. With the increasing use of prefetching [10], the traffic optimization between cache memories and main memory becomes crucial. The cache memory traffic depends

Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)
1550-6207/05 $ 20.00 **IEEE**

mainly on the cache write policies, the write through policy, and the write back policy, each of which has its own merits and demerits.

The write back policy, the one widely used, tries to minimize the bandwidth requirement, by writing back the dirty blocks only at replacement and hence multiple cache writes may result in a single memory write. Moreover, the write instruction is executed at cache speed, thus it is fast. However, the write back policy results in slow context switching, in case of multiprogramming environment. Furthermore, the memory is not always consistent with the cache, which is important, not only in multiprocessor systems, but also in single-processor systems as well. For example, when some DMA devices make checks on the memory. Finally, a read miss to the cache may cause a write to the main memory, or to the next level cache.

On the other hand, the write through policy makes the cache always consistent with the memory. The read miss never results in write operation. Furthermore, the write through policy is easy to implement. However, the bandwidth requirement is huge, which can lead to high power consumption as well as more severe bus contention. Another important drawback of the write through policy, is that the write operation itself is slow. This is mainly due to write buffer overflow.

In the writeback scheme, if the system bus is congested, the buffer gets filled quickly, leading to loss in performance. The system bus is not only used by the processor, but it is also used by some DMA devices, and some devices, like the graphics accelerators, thus leading to congestion. Hence, it is apparent that we need a new policy that can make better use of the bandwidth, while maintaining the advantages of both the schemes.

In this paper, we present a technique that combines both the compiler technology with its capability to analyze the full program, and the microarchitectural technique, which can *see* the runtime behavior. We try to optimize the cache-to-memory traffic in two ways:

- Making use of the idle bus cycles to send back the dirty blocks to memory. Hence optimizing the traffic by better *bandwidth management*. This depends on the dynamic behavior of the program, therefore, will be done by the microarchitecture.
- Using the compiler to find the dead values. These values are not written back to memory, because they will not be referenced again. Therefore, we optimize the traffic using *bandwidth saving*. This requires a global view of the program, thus, is done by the static analysis and is conveyed to the hardware by the compiler.

For bandwidth management, the hardware predicts the last store to a block before replacement. Using this information, the block is written back to the memory or the next level cache, when the bus is not heavily used. This does not happen in the critical path and therefore does not affect the overall performance. The bandwidth saving is accomplished through static analysis, by determining that a specific value will be dead after a certain instruction, hence will not need to be written back in case of replacement.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work. The bandwidth management technique is presented in Section 3, while the bandwidth saving techniques are discussed in Section 4. Experimental results are shown in Section 5, followed by discussion. Finally, Section 6 concludes and summarizes the paper.

2 Related Work

Increasing hit rate and reducing miss penalty have been the main paths taken by the researchers to improve the cache performance. Victim cache [13] is one of the earliest attempts to do that, followed by many improvements [1][7]. The technique presented in [18] showed a way of using the holes in the direct-mapped cache to decrease conflict misses. In [17], a technique for predicting a miss and aborting the operation is proposed. [19] proposed partitioning the first level data cache for clustered microarchitectures, to be able to provide the timely bandwidth required with the increased frequency. The effects of long

memory latencies and increased memory bandwidth requirements on the design of modern microprocessors and their memory systems have been discussed in [5].

Many studies have focused on compiler analysis and optimization to improve the cache performance [8][12][11][9]. All these proposed techniques try to reduce cache misses by improving data locality. The compiler does so by either placing the data efficiently in memory [8][12][11], or change the memory access order to improve the temporal and spatial locality.

Some work has been done in dead value detection for register values [6] [16]. [6] proposed a hardware technique for detecting dynamic instruction instances that generate unused results in registers. [16] performs static analysis to determine the dead register information, and uses this information at runtime to perform various optimization, such as decreasing the physical register file size, and eliminating register save and restore, at procedure calls and context switches. Using dead block prediction to enhance prefetching is proposed in [14].

Improving bandwidth utilization has been proposed in [15], where the cache lines that have been marked dirty and become the least recently used state are written back to the memory or the next cache level earlier than in a conventional writeback policy. However, this technique cannot be directly applied to a direct mapped cache for example, or with a cache that used another replacement policy. The hybrid technique we propose in this paper, can be applied to any cache configuration.

3 Bandwidth Management

The main idea of bandwidth management is to predict the number of writes to a cache block before a replacement. When this number is reached, the block is written back to the main memory or the next level cache, if it is dirty, without waiting for the block to be replaced. Hence, when the time comes for replacing the block, it will be clean and the replacement will be done fast. The whole operation is done outside the critical path, hence the overall system performance is not affected. Besides, a misprediction of the number of writes, will result only in a small increase in the number of writes to the memory or the cache. We found that this small increase is much smaller than a write through.

The method consists of augmenting each cache slot with two saturating counters and one bit. The first counter, called *current*, counts the number of writes to that block. The other counter, called *predictor* contains the number of writes expected to that block. The *written* bit is set if the block is written back to the memory of next level cache *before* it is replaced. If the *predictor* counter contains zero, then the conventional write back policy is used. Whenever a store is done to that block, the *current* counter associated with that block is incremented. When the *current* counter reaches the predicted number, that is, the value in the *predictor* counter, the block is written back to the memory, or next level cache, the *current* counter is reset, and the *written* bit is set.

When a block is to be replaced, the *written* bit is checked. If it is set and the *current* counter is non-zero, the *predictor* counter is incremented, and the block is written back to memory. If the *written* bit is not set and the *current* counter is non-zero, the *predictor* counter is decremented. After the update of the *predictor* counter, both the *written* bit and the *current* counter are reset.

4 Bandwidth Saving

In this section we discuss two compiler mechanisms that save the memory bandwidth by detecting redundant writes during cache block replacement. In the dead value detection (DVD) mechanism, the compiler detects the memory locations whose values are dead and therefore, there is no need for writing those values to the higher level when replacing the corresponding cache lines. The dead stack detection (DSD) mechanism detects the dead stack locations in the cache and cleans those lines on procedure return.

4.1 Dead Value Detection

Cache block replacement can be optimized by using the dead value information provided by the com-

/* Assume variable X is in location 20(SP),
and Y is is 40(GP) and all the accesses of X and Y are shown */

(a)	(b)
...	...
I_{10} : $sw\ \ r1, 20(SP)$ /* X is live after I_{10} */	I_{10} : $sw\ \ r1, 20(SP)$
...	...
I_{20} : $lw\ \ r3, 20(SP)$ /* X is live */	I_{20} : $lw\ \ r3, 20(SP)$
...	...
I_{30} : $lw\ \ r4, 20(SP)$ /* X is dead after I_{30} */	I_{30} : **lw.last** $r4, 20(SP)$
...	...
I_{100} : $sw\ \ r5, 20(SP)$ /* X is live after I_{100}*/	I_{100} : $sw\ \ r5, 20(SP)$
...	...
I_{120} : $sw\ \ r0, 40(GP)$ /* Y is dead after I_{120} */	I_{120} : **sw.last** $r0, 40(GP)$
...	...
I_{140} : $sw\ \ r5, 40(GP)$	I_{140} : $sw\ \ r5, 40(GP)$
...	...

Figure 1: Example code to show the cache optimization using dead value information

piler. The existing write back cache replacement policy writes the contents of a cache block into the next memory level if the block being replaced is dirty. However, it is possible that the value stored in that cache block is no longer needed, i.e. its last use has already taken place. In such a situation we can just replace that dirty block without writing back the dirty block.

Consider the example shown in Figure 1(a). A value is stored into location X by instruction I_1. Instructions I_{20} and I_{30} read the value from location `X` and use it. Afterward, instruction I_{100} stores a new value in `X` and then instruction I_{120} reads it again. During program execution, the cache line holding the location `X` will be marked dirty after I_1. If this line is replaced before I_{30} then the line has to be written back to memory. However, if the replacement occurs after I_{30} and before I_{100}, then although the line is dirty, there is no need to write the data back into the memory, i.e.location `X` has become dead after I_{30}. `X` becomes live again with the write by instruction I_{100}.

This is a very common scenario in a program, because most variables (memory locations) go through a cycle of *write*, one or more *reads*, followed by another *write*, and so on. Sometimes there are multiple *writes* to a memory location without any intervening reads [1]. Furthermore, lots of memory locations become dead because of register spilling, where a value is temporarily stored in memory from a register and then loaded again from memory to the register just once.

Compiler can easily detect the scenarios mentioned above in a program by using standard data flow techniques. We have developed a compiler algorithm[3] to detect the dead value information. This is used by the cache to optimize the replacement policy. The compiler detects the program points where a certain value becomes dead and passes that information to the processor. A cache line becomes dead when all the bytes in the cache line are either clean or dead. We have also developed the mechanism and hardware support needed to pass the dead value information from the compiler to the processor.

4.1.1 Overview of Compiler Algorithm for Dead Value Detection (DVD)

In this sub-section we give a brief overview of our compiler algorithm for the dead value detection. The details of the algorithm can be found in [3]. Our DVD compiler works as a post-link optimizing compiler.

[1] Ideally these useless stores should be determined and eliminated by the compiler using dead code elimination

To make the analysis safe, the compiler algorithm assumes all variables to be live unless known for sure that the variable is dead.

In our compiler algorithm, we have assumed that all memory locations are accessed using base addressing mode, i.e. a memory location is specified by adding an offset to the base pointer. In our compiler, we have only identified the dead values for memory locations accessed by three base pointers - global pointer, stack pointer, and frame pointer, i.e., through base registers $r28, $r29, $r30 respectively following MIPS register usage convention.

The algorithm starts by building the *control flow graph (CFG)* of the current procedure. During the CFG building phase, the compiler also creates the local and global symbol tables by tracking the local and the global variables, identified by the offset and the base pointer through which they are accessed. The offsets of local variables are constant throughout the procedure since the value of the base pointers do not change inside the procedure. Similarly, the offsets of the global variables are constant throughout the program. Our compiler algorithm takes care of the situation where a memory location is accessed through a base register other than the stack or the global pointer. After creating the local and global symbol tables, the *read* and *write sets* [2] are generated for every instruction and then the dead value information are computed. A variable x is dead after an instruction *inst* reading (or writing into) x in block B, if *inst* is the last instruction reading(or writing) x in B and x is not live at the exit of B. Also x is dead after instruction *inst*, if there is an instruction writing to x in B before any other instruction reading x.

The dead value information is passed from the compiler to the processor by modifying the load and store instructions. Note that a value in a memory location can be dead only after a load or a store instruction that accesses that particular memory location. If a load instructions loads from a memory location after which that value in that location becomes dead then the compiler changes that load instruction to *load.last*. Similarly after storing a value in a location, if there is no use of that value then the compiler changes that store instruction to *store.last*. Figure 1(b) shows the code after compiler transforms *lw* and *sw* instructions (after which the corresponding memory locations are dead), to *lw.last* and *sw.last*.

[2] *read* and *write* sets contain the set of variables read and written by an instruction respectively.

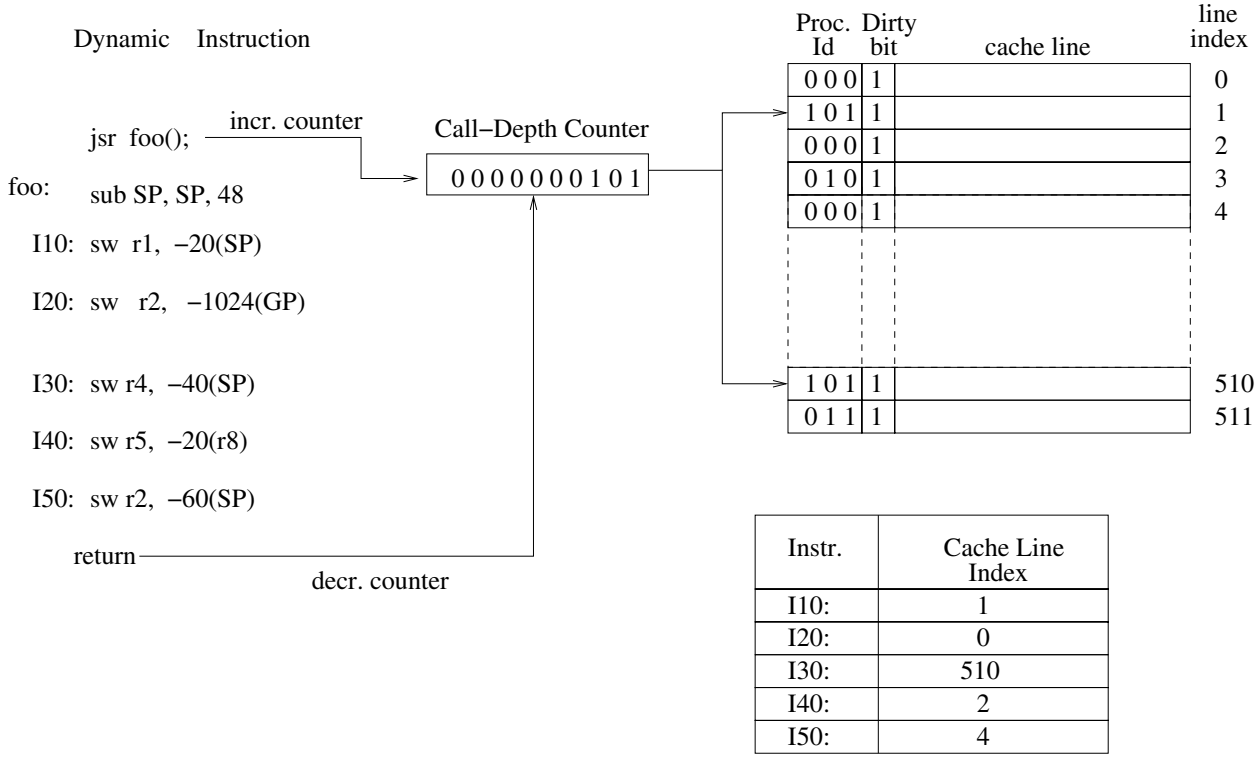
4.1.2 The Hardware Interface for Dead Value Detection

In this subsection we briefly describe the hardware required to process the dead value information obtained from the compiler. The details are in [3]. The compiler passes the dead value information at the word and double-word level because most of the memory accesses are done at that granularity. To maintain the dirty/clean information of the cache lines at the word level granularity. we use a separate small table called dead entry table (DET), otherwise the overhead would be too high. Each entry in the table contains the block address and an n ($n = cache\_line\_size/word\_size$) bit flag for dirty/clean information and one valid bit.

Initially DET is empty with all the valid bits reset. On a cache write an entry is allocated in DET (if not already present). When a new entry is allocated, all n bits in the flag are set if the cache line is dirty. Otherwise, all the bits in the flag are reset. With subsequent writes the corresponding bits are set. On *lw.last* or *sw.last* if the entry is already in DET, the corresponding bit is cleared. Also *sw.last* does not set the dirty bit in cache. When all the n bits in a DET entry are 0, the processor resets the dirty bit of the cache line. Hence, this cache line will not be written back during replacement. To allocate a new entry in the DET, we first look for an entry whose valid bit is reset. Otherwise randomly replace an entry.

4.2 Dead Stack Detection (DSD)

In the programs written in languages like C, the stack of the procedure holds the data local to that procedure and the lifetime of the stack variables are only limited to the lifetime of the procedures. A procedure allocates a stack in the memory when it is instantiated, by decreasing the stack pointer. Just before



Instr.	Cache Line Index
I10:	1
I20:	0
I30:	510
I40:	2
I50:	4

Figure 2: An example showing the cleaning of cache lines accessed by a procedure stack

a procedure returns, it deallocates the stack by incrementing the stack pointer and the local variables in the stack becomes dead after that procedure returns. Hence any update made to the cache lines corresponding to a procedure's stack need not be propagated to the next memory level after that procedure returns, because the value in that stack will not be used again. Therefore, we can safely make all the lines in the cache clean, corresponding to a procedure's stack at the return of that procedure. This can easily be done by the processor with the help from the compiler.

We have developed compiler and hardware method for dead stack detection [3]. By default, the processor assumes that all the cache lines that are accessed due to a memory access through stack pointer would be dead after the accessing procedure returns. However, sometimes the procedure accesses its caller's stack by using the stack pointer. These lines should not be marked clean by the processor at the return of the procedure. During compilation the compiler can easily detect such accesses. If a stack access uses an index value greater than the size of its local stack, then compiler annotates that access as a *global access* and the processor does not clean that line upon procedure return. In our compiler, such *global accesses* are identified along with the DVD phase.

4.2.1 Overview of the Hardware Support for Dead Stack Detection

In order to clean a line at a procedure return, the processor needs to uniquely identify the lines accessed during stack access. To do that we have introduced the concept of ownership of a cache line. A cache line is either owned by the global area or by a dynamic instance of a procedure. We have added four bits to each cache line to store the owner's identifier (id). A cache line is cleaned when its owner procedure returns.

Call depth is used to identify a dynamic instance of a procedure because at any one point of time there is only one procedure at a certain call depth. The global area is at depth 0 and procedure *main()* of a program is at depth 1 and so on. We have used a special 4-bit counter called *call_depth_counter* for this purpose. At the start of a program execution, *call_depth_counter* is initialized to 1. During execution, when a procedure call is encountered, the *call_depth_counter* is incremented by 1. Similarly at procedure return the *call_depth_counter* is decremented by 1.

When a load or store instruction, using stack pointer, is executed by the processor, and the instruction is not annotated as a *global access* by the compiler, the processor sends the accessing procedures id to the cache along with the other information. Otherwise the ownership id value of 0 is sent to the cache to denote it as an access to the global area. [3]

If the access results in a cache miss, then the procedure id is written in the ownership field of the cache line. In case of a cache hit, if the ownership value in the cache line is less than the ownership id sent to the cache, then the existing ownership of that line is not changed. Because the line is already owned by the global area or by a procedure with less depth (i.e. higher in the calling chain) and the line should be cleaned only when that procedure returns (which would be later than the return of the current procedure). Otherwise the ownership field is updated with the procedure id.

At procedure return, all the lines owned by the returning procedure are marked clean. This is done in a linear scan through the whole cache and since this is outside the critical path of the processor, it will not affect the cache access time or cycle time. Also in a linear scan, the hardware cost is not much. In the worst case, some line may get replaced or ownership may get changed before the line is cleaned. This will not affect the correctness of the execution.

4.2.2 Example of Dead Stack Detection (DSD)

The key concepts of DSD is explained with the example shown in Figure 2. In the left of Figure 2, a segment of dynamic instruction stream in shown. The program enters procedure *foo()* with the *jsr* instruction. We only show the store instructions in

[3]Since we have only used 4 bits in the cache line ownership field and there are 16 bits in the *call_depth_counter*, if the counter value is greater than 15, the value 15 is passed to the cache as the ownership id.

procedure *foo()*. In the right of the Figure 2 we show the state of the cache just before returning from *foo()* and the table below maps the *sw* instructions to the cache lines.

Here we assume all the *sw* accesses were cache misses, and lines were allocated for them. Assume that before the call instruction, the depth-counter value was 4. At *jsr foo()*, the depth-counter is incremented to 5. So the identifier of the dynamic instance of procedure *foo()* under consideration is 5.

When I10 accesses offset 20 in its activation stack, it is loaded in cache line index 1. Hardware can determine that the access is made to the local stack of *foo()*, because stack pointer SP is used as the base pointer and offset 20 is less than the stack size of *foo()*, which is 48. So the call-depth counter value 5 is stored in *Proc ID* field of cache line 1. Similarly, for I30 also 5 is stored in the *Proc ID* field of cache line 510.

However, I20 is accessing a global value, since accessed through global pointer, 0 is stored in the *Proc Id* of cache line 0. Similarly, 0 will be stored in the *Proc Id* field of cache lines 2 and 4 for accesses made by I40 and I50. Because we do not know the value of r8 and therefore do not know whether local stack is accessed or not.[4]

I50 is accessing beyond the local stack. Therefore, cache line 4 also should not be made clean after the procedure return. Now at the return instruction, the dirty bits of lines 1 and 510 are reset. The local values of the instance of *foo()* are dead after the return of *foo()* and there is no need to store them to memory for future use.

If for example, the access made by I30 shares the cache line with its caller and the line was already present in the cache with the *Proc ID* value of 4, (the caller's depth), then *Proc Id* field would not have been overwritten by I30. This is because we do not want to discard the changes made by the caller. So, in general, the *Proc Id* field of an existing cache line will not be overwritten by a larger depth value. But it will be overwritten by a smaller depth value.

[4]It may be possible to know where r8 is pointing with better link time compiler analysis.

Processor Params	Value
Decode Width	4
Issue width	4
Branch Predictor	Bimodal with 2048 table size
L1 - Icache	32KB, 4-way set assoc., LRU, 32 byte line size, 1 cycle latency
L1 - Dcache	32KB, 4-way set assoc., LRU, 32 byte line size, 1 cycle latency
L2 - Unified	256KB, 4-way set assoc., LRU 64 byte line size, 6 cycle latency
DET	1KB
Memory Latency	100 cycles for the first chunk

Table 1: Simplescalar Simulator Parameters

Integer Benchmark	Number of References	FP Benchmark	Number of References
bzip2	235864202	*ammp*	255247899
gcc	389188032	*apsi*	187965389
gzip	150676316	*art*	212728035
mcf	282954558	*equake*	161264237
perl	248595729	*mesa*	249097078
twolf	254806912	*swim*	136862642
vortex	275013769	*wupwise*	175958198
vpr	214718042		

Table 2: Total Number of Loads and Stores Committed

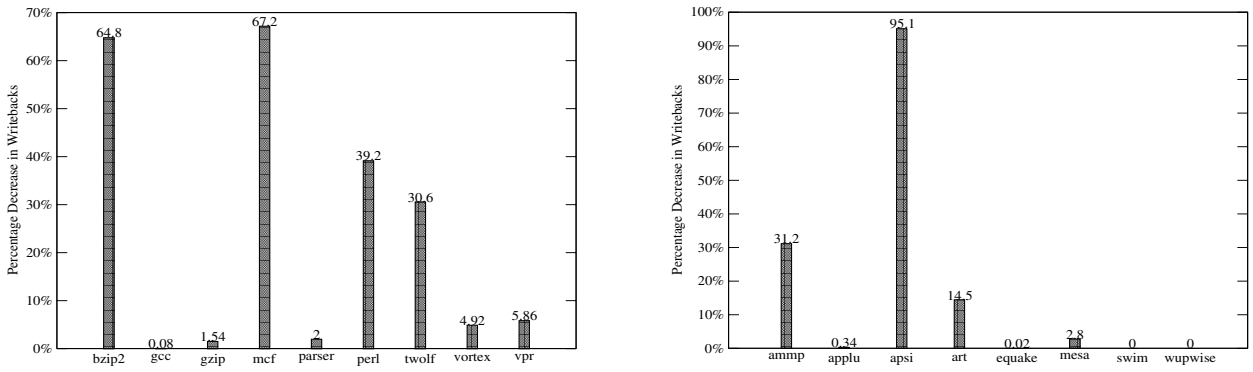


Figure 3: Percentage Decrease in Writebacks in L1 Data Cache

5 Experimental Evaluation

In this section we discuss some experimental results in bandwidth management as well as bandwidth saving.

5.1 Experimental Methodology and Setup

For microarchitectural simulations, we modified the out-of-order processor simulator of the Simplescalar tool set [4], with PISA (portable ISA) instruction set. Table 1 shows the parameters of the simulator.

We used the integer and floating point benchmarks from SpecCPU2000 suite with reference input. The benchmarks have been compiled using the Simplescalar `gcc` with the optimizations specified in the `makefile` provided with the suite. Each benchmark is simulated for 500M instructions after skipping the startup phase as indicted in [20]. Table 2 shows the total number of loads and stores committed. The counters are saturating counters of 5 bits each. The bandwidth saving techniques are applied to the data cache closest to the processor. On the other hand, the bandwidth management technique is done to the cache farthest from the processor.

5.2 Experiments and Discussion

The first set of experiments shows the percentage decrease in writebacks in L1 data cache. This is an indication of the number of writes to the memory cleared by the compiler, hence saving bandwidth. Figure 3 shows the results for both SpecINT and SpecFP. The results are shown with respect to the writebacks in the conventional write back policy of the cache. As can be seen from the figure, the savings are higher in SpecINT (average of 24.02%) than in SpecFP (average of 17.99%). This is mainly due to the complex control flow in the integer programs that increases dead values generated by local variables in subroutine calls. The SpecFP benchmarks on the other hand, have simple control flow, consisting mainly of loops in most cases. The only exception is `apsi` where almost 95% of the writebacks are eliminated. On the other extreme, `swim` and `wupwise` have not gained from the bandwidth saving techniques. It is to be noted that the IPC (instruction per cycle) is slightly better for the schemes with the bandwidth reduction (around 5% improvement for specINT and 3% improvement for SpecFP on average), this is mainly due to the fact that we have not yet modeled the bus congestion in our simulator, and the gain in IPC seen is due to the decrease in the traffic which reduces write buffer full

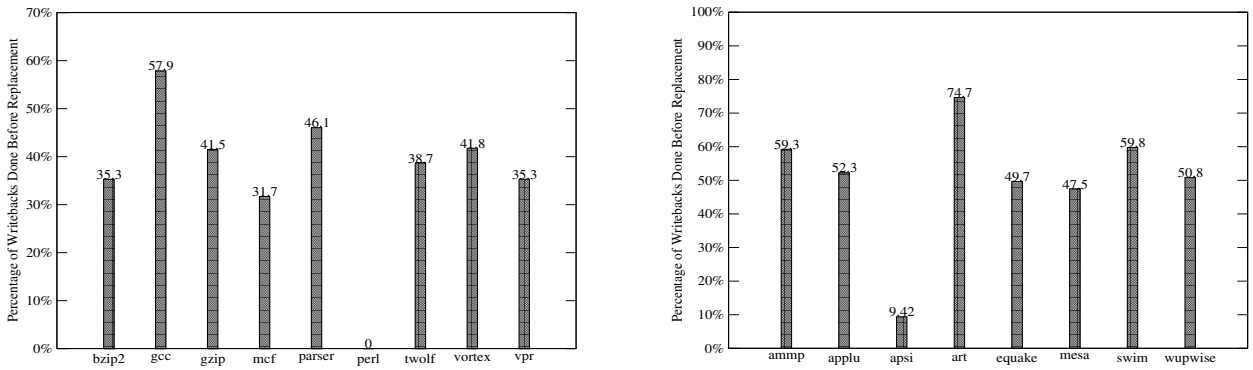


Figure 4: Percentage Dirty Blocks Cleared Before the Block is Replaced in L2 Unified Cache

scenario. hence we do not report them here. With bus congestion modeled, further IPC enhancement is expected to be seen. The writebacks from the unified L2 to the main memory is also reduced, because the write requests received from L1 is already reduced. These reduced writebacks at L2 are further optimized by the bandwidth management techniques

The second part of the traffic optimization is the bandwidth management from the unified L2 to the main memory. In this case we do not decrease the number of writebacks, but we try to re-distribute the writebacks over time in order to make use of the low traffic periods on the bus. Therefore a good metric here is the percentage of writebacks sent to memory *before* the block is actually replaced. This is shown in Figure 4. As can be seen, more than half the writes to L2 in SpecFP and almost half of the writes to L2 in SpecINT are done during low traffic times, before the block is replaced. `perl` is the only benchmark which did not benefit from this technique. This is because the original number of writes sent from L2 to the main memory is very small (there is only 0.0214 average writes to a block before being replaced).

5.3 Results of Bandwidth Savings Techniques

In this subsection we report the results for the compiler based techniques alone without any bandwidth management techniques. In Table 3, we show the percentage of times a dirty cache line is cleaned by using either or both of the compiler proposed methods. Column two contains total number of writes into L1 cache during the program execution without any of the optimization. The percentage of times a dirty cache line has been cleaned by DVD, DSD, and both are shown in the table. We further break down the combined results into contribution from DVD and DSD. The results are obtained using a 64 entry DET.

From the table 3 we see that in most benchmarks the performance of DVD is much better than that of DSD. Because DSD is much conservative. All stack accesses through other registers are considered as global area. Also if the line is shared between a caller and callee, it is not cleaned before caller returns, by that time the line may have already been replaced. More detailed register alias analysis could give better improvement for DSD.

DVD is showing good performance. This implies that the compiler could identify the dead values properly and could be effectively used for reducing cache

Suite	Benchmarks	# of Writes	% of Dirty lines cleaned			
			DVD	DSD	DVD + DSD	
SPECINT	*bzip2*	54233795	4.11	2.03	4.11	1.82
	gcc	215145717	0.05	0.07	0.05	0.09
	gzip	44842012	14.68	4.72	14.28	0.06
	mcf	64138843	5.21	7.52	5.21	2.33
	parser	72153638	15.19	5.92	15.04	0.99
	perl	73879261	14.82	12.76	14.82	3.71
	twolf	56590048	6.47	3.62	6.39	1.57
	vortex	116586943	6.11	0.17	6.10	0.06
	vpr	45439516	5.39	3.88	5.38	0.55
SPECFP	*ammp*	50857335	77.35	1.46	77.32	1.19
	applu	47762212	0.25	0.15	0.25	0.06
	apsi	72486228	19.58	6.54	19.24	6.40
	art	44018039	41.36	0.41	41.36	0.41
	equake	42113970	33.19	0.10	33.19	0.07
	mesa	69344655	28.67	3.22	28.31	2.95
	swim	40628454	22.22	11.11	22.22	11.03
	wupwise	62717771	34.87	2.78	34.87	1.04

Table 3: Statistics for Cleaning Dead lines for DVD and DSD

traffic. DVD could identify the dead global variables that DSD cannot. In case of combined scheme, DVD cleans most of the lines before DSD could get a chance to clean them. Therefore we see that in most of the cases in the combined scheme DVD retains its performance.

6 Conclusions

In this paper, we have discussed some techniques for optimizing the traffic between different levels of cache memories, as well as between the cache memory and the main memory. We have shown that by combining both compiler techniques and dynamic method, we can achieve both bandwidth saving as well as better bandwidth management. Compiler techniques have been able to remove 24% of the writes for the SpecINT and 18% of the writes for the SpecFP. Moreover, the bandwidth management was able to write around 50% of the writebacks during the low traffic periods before the block is actually replaced.

The future work includes the study of the effect of the proposed hybrid scheme on the traffic by simulating the bus congestion, as well as with more cache levels. In case of more than two level caches, the bandwidth saving techniques need to be applied to the cache nearest to the processor, while the bandwidth management technique can be applied to all other levels, or at least to the level nearest to the main memory.

References

[1] A. Agarwal and S. D. Pudar. Column-associative cache: A technique for reducing the miss rate of direct-mapped caches. In *Proc. 20th Int'l Symposium on Computer Architecture*, pages 179–190, 1993.

[2] Semiconductor Industry Association. International technology roadmap for semiconductors, 2001.

[3] A. Bhowmik and M. Zahran. Cache traffic optimization. Technical Report IISc-CSA-TR-2005-1, Computer Science and Automation, Indian Institute of Science, India, January 2005. URL: http://archive.csa.iisc.ernet.in/TR/2005/1/.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS TR-1308, University of Wisconsin Madison, July 1996.

[5] D. Burger, J. R. Goodman, and A. Kgi. Limited bandwidth to affect processor design. *IEEE Micro*, 17(6):55–62, 1997.

[6] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 2002.

[7] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proc. 2nd Int'l Symposium on High Performance Computer Architecture*, 1996.

[8] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1998.

[9] S. Carr, K. McKinley, and C. W. Tseng. Compiler optimizations for improving data locality. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1994.

[10] T-F Chen and J-L Baer. A performance study of software and hardware data prefetching schemes. In *Proc. 21st Int'l Symposium on Computer Architecture*, 1994.

[11] T. Chilimbi, B. Davidson, and J. Larus. Cache conscious structure definition. In *Proc. of Int'l Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[12] T. Chilimbi, M. Hill, and J.Larus. Cache conscious structure layout. In *Proc. of Int'l Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[13] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffer. In *Proc. 17th Int'l Symposium on Computer Architecture*, pages 364–373, May 1990.

[14] A-C Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proc. Int'l Symposium on Computer Architecture (ISCA) 24*, 2001.

[15] H-H S. Lee, G. S. Tyson, and M. T. Farrens. Eager writeback- a technique for improving bandwith utilization. In *33rd annual IEEE/ACM international symposium on Microarchitecture*, December 2000.

[16] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proc. of International Symposium on Microarchitecture (MICRO-30)*, 1997.

[17] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Proc. 9th Int'l Symposium on High Performance Computer Architecture*, 2003.

[18] J-K Peir, Y. Lee, and W Hsu. Capturing dynamic memory reference behavior with adaptive cache toplogy. In *Proc. Int'l Conf. on Architecture Support for Programming Languages and Operating System (ASPLOS)*, 1998.

[19] P. Racunas and Y. N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *Proceedings of the 17th international conference on Supercomputing*, pages 22–31, 2003.

[20] S. Sair and M. Charney. Memory behavior of the spec2000 benchmark suite. Technical Report RC-21852, IBM T. J. Watson Research Center, October 2000.

[21] D. Weiss, J Wuu, and V. Chin. The On-Chip 3-MB Subarray-Based Third-Level Cache on an Itanium Microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11), 2002.