Cache Traffic Optimization

Anasua Bhowmik Mohamed Zahran

IISc-CSA-TR-2005-1
http://archive.csa.iisc.ernet.in/TR/2005/1/

Computer Science and Automation
Indian Institute of Science, India

January 2005

Cache Traffic Optimization

Anasua Bhowmik[*]
Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012, India
anasua@csa.iisc.ernet.in

Mohamed Zahran
Dept. of Electrical Engg.
City College
City University of New York
New York, NY 10031, USA
mzahran@ccny.cuny.edu

Abstract

The center of gravity of computer architecture is moving toward memory systems. This is due to many factors. First, one of the main performance bottlenecks is the gap between memory system and processor. This results in many moving as much memory as we can from off-chip to on-chip. Three levels of cache memory on-chip is not uncommon nowadays. Furthermore, we are on a sustained effort into integrating a larger number of devices per chip. This renders integrating a large memory on-chip a reality.

Although the enabling technology allows larger caches to be embedded on-chip, these caches are not giving the required performance. One of the main reasons for that, is the delay in writing back the data of the replaced block to memory or to the lower level cache. This makes the block replacement time consuming, and hence affects the overall performance.

In this paper, we present several techniques to enhance the cache traffic problem. The first technique, called *lazy-write*, predicts the time at which the cache block will no longer be written before replacement, and write it back to the memory, if it is dirty, at time of low traffic. Hence, when the block is being replaced, it will be clean and the replacement will be done much faster. The other technique, deal with detecting values that are *dead*, and hence do not need to be written to the memory altogether. Therefore, reducing the traffic to the memory and make the replacement faster. Hence, two main contributions are presented in this paper. The first is traffic optimization using bandwidth management. The second is traffic optimization using bandwidth saving.

1 Introduction

Memory systems continue to present a major hazard in our quest to improve processor performance. Cache memories have received great attention as they represented the

[*]Author for TR correspondence. `anasua@csa.iisc.ernet.in`

winning horse in decreasing the gap between the processor and the memory subsystem.

The advances in process technology have resulted in a huge transistor budget available per chip. One of the main uses of this huge budget is to have several levels of cache memory on-chip.

The Intel Itanium processor is an example of a state-of-the-art microprocessor, which has 3 on-chip cache levels, for a total of 3 MB [18]. Level 3 cache is taking about 43% of the die area. This means that future processor chips will be nothing but a big, fat cache memory attached to a small CPU!

As Moore's law continues to be valid, the semiconductor industry association (SIA) predicts that process technology advancement will continue for a decade at least, until we hit a physical limit [3].

However, the cache memory performance is not delivering the required performance. This is due to many reasons, such the changing cache requirements (block size, associativity, etc.) across applications, or throughout the lifetime of a single application. One important reason which prohibits the increase in cache performance, is the traffic between a cache in a certain level and the cache in the level below, or the main memory. With the increasing use of prefetching [9], the traffic optimization between cache memories and main memories becomes crucial. The cache memory traffic depends on two main policies, the write through policy, and the write back policy. Each one of these policies has its own pros and cons.

The write back policy, which is the one widely used, is a less memory intensive. That is, it tries to minimize the bandwidth requirement, by writing back the dirty blocks only at replacement. This means multiple cache writes may result in a single memory write. Moreover, the write is done 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 in single-processor systems also, when some DMA devices make checks on the memory. Finally, a single read miss to the cache may cause a write to the main memory, or to the lower 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 power consumption as well as bus contention. Another important drawback of the write through policy, is that the write operation itself is slow. This is due mainly to write buffer overflow.

Figure 1 shows the percentage of write accesses where the write buffer was full and causes delay. We have made this experiment using a write buffer of size 4. As can be seen from the figure, more than half the time the write buffer is full. In the writeback scheme, if the system bus is congested, the buffer gets filled quickly, leading to loss in performance. The system bus can get congested because it is not used by the processor alone, some DMA devices, and some devices like the graphics adapter use the bus also, leading to congestion.

Hence, it is apparent that we need a new policy that can make better use of the

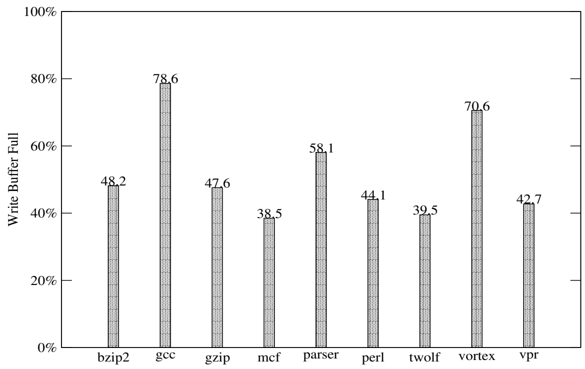


Figure 1: Percentage of Write Operations Facing Write Buffer Full Scenario in Write through Policy

bandwidth, while maintaining the advantages of both schemes.

In this paper, we present two main techniques for enhancing the cache memory traffic. We try to optimize the cache-to-memory traffic using two ways:

- Making use of the idle bus cycles to send back the dirty blocks to memory. Hence optimizing the traffic by better *b*andwidth management.

- Using the compiler to find the dead values and hence they are not written back to memory, since they will not be referenced again. Therefore, optimizing the traffic using *b*andwidth saving.

The first technique, called *lazy-write*, predicts when a block will not be written again before replacement. Using this information, the block is written back to the memory of the lower level cache when the bus is not heavily used, and this does not happen in the critical path, so it does not affect the overall performance. The second technique, called *dead-value-detection (DVD)* predicts 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 first technique, called lazy-write is presented in Section 3. The dead value detection technique is discussed in Section 4. Experimental results are shown in Section 6, followed by discussion. Finally, Section 7 concludes and summarizes the paper.

2 Related Work

Increasing hit rate or reducing miss penalty has been the main path taken by researchers to deliver the best performance. Victim cache [12] is one of the earliest attempts to do that, followed by many improvements, for example [1][6]. The authors in [15] in proposed a way of using the holes in the direct-mapped cache to decrease conflict misses. In [14], the authors predict a miss and abort the operation. In [16], the authors proposed to partition the first level data cache for clustered microarchitectures, in order to be able to provide the timely bandwidth required with the increased frequency.

Some work has been done in dead value detection for register values [5], by detecting dynamic instruction instances that generate unused results.

The live variable analysis is a very useful technique used for various static optimizations like register allocation, dead code elimination [2]. However, we do not know of any work that passes the compiler generated dead value information to the hardware for dynamic optimization. During register allocation, whenever a new register is needed, the allocator tries to get a register containing dead value. Because, then there is no need to write the content of the register back to the memory. In dead code elimination, the compiler determines whether the value computed by an instruction will be used in future, i.e. whether the instruction is generating a dead value or not. If so, then the instruction producing the dead value can be eliminated by the compiler.

There have been many works [7, 11, 10, 8] that focus on compiler analysis and optimization to improve the cache performance. All these works basically try to reduce cache misses by improving data locality. The compiler does so by either by placing the data efficiently in memory [7, 11, 10] or change in the memory access order to improve the temporal and spatial locality.

Improving bandwidth utilization has been proposed in [13], 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 Lazy-write

The main idea about lazy write 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 lower 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 that a write through. We have implemented this idea in two different methods.

The first method depends on a *Lazy Write Table (LWT)*. Figure 2 shows the design. When a block is accessed, the LWT is interrogated to predict the number of writes to that block before it gets replaced. If no prediction is given, then the conventional write back policy is used. Whenever a write is done to that block, a counter associated with that block is incremented. When the counter reaches the predicted number, the block is

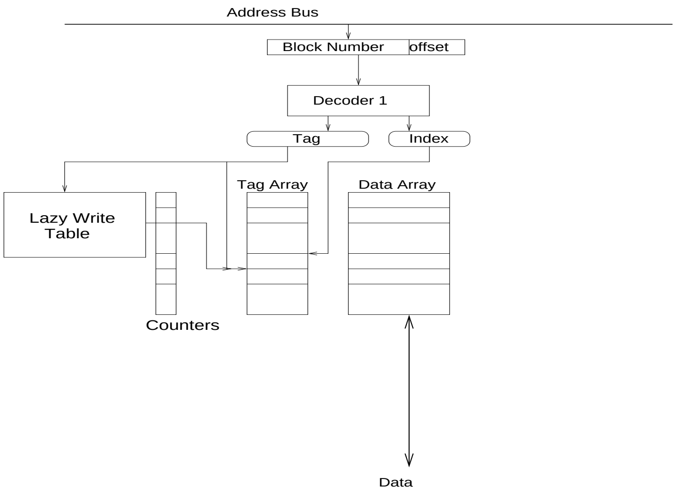


Figure 2: Lazy Write Using LWT Table (Address Wise Method)

written back to the memory, or lower level cache. If the block is not replaced, and extra writes have been done to it, then at the time of its replacement, the new count is written to the LWT table and the counter is reset. The LWT table is a small cache, We call this method *address wise (AW)*, because the prediction is done based on the address of the block.

Another simpler method is the *block wise (BW)*. In BW, instead of keeping the prediction in a separate table, each cache slot is associated with two counters. The first counter counts the number of writes to that block. The other counter contains the last number of writes to that block before a replacement. Hence the counters are associated with he block or cache slot, instead of being in a separate table.

4 Dead Value Detection

Cache block replacement can be further optimized by using the dead value information provided by the compiler. The existing write back cache replacement policy writes the contents of a cache block into the higher level of memory 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 usage has already taken place. In such a situation we can just replace that dirty block with another block without writing the dirty block into higher level.

Consider the example shown in Figure 3. 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}.

$I_1 : WRITE\ \ X$ /* X is live after I_1 */
...
$I_{20} : READ\ \ X$ /* X is live */
...
$I_{30} : READ\ \ X$ /* X is dead after I_{30} */
...
$I_{100} : WRITE\ \ X$ /* X is live after I_{100}*/
...
$I_{120} : READ\ \ X$

Figure 3: Example code to show the cache optimization using dead value information

This is a very common scenario in a program since 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]. Also lots of memory locations become dead because of register spilling, where a value is temporarily stored in memory from 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. In this paper, we present the compiler algorithm to detect the dead value information. This is used by the cache to optimize the replacement policy further. The compiler detects the program points where a certain value becomes dead and passes the information to the processor. A cache line becomes dead when all the bytes in the cache line are either clean or dead. The mechanism and the hardware support needed to pass the dead value information from the compiler to the processor is discussed in subsection 4.2.

4.1 Compiler Algorithm for Dead Value Detection

In this sub-section we describe the compiler algorithm for the dead value detection. We have implemented the compiler algorithm on the program binary. 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. This is indeed the case for the binaries generated by the PISA compiler. 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. Stack pointers and frame pointer values are generally computed once at the beginning of the procedure at the beginning and kept constant till the end of the procedure. Similarly, the global pointer is assigned the value only once at the beginning of the program and it remains unchanged till the end of the program. For all other registers used as base pointer we have adopted

[1]Ideally the compiler should remove all the unnecessary stores through dead code elimination

a conservative approach.

The overview of the algorithm is given in Algorithm 1. The *dead_value_detection()* algorithm works on one procedure at a time. There is no inter-procedural analysis. The algorithm starts by building the *control flow graph (CFG)* [2] of the current procedure. The procedure *compute_register_aliases()* finds out whether a register *$r* is accessing the same memory area as pointed to by stack pointer, global pointer or frame pointer. To make the analysis safe, the compiler algorithm must assume all variables to be live unless known for sure that the variable is dead. Therefore, if a load instruction uses register *$r* as the base pointer, then we need to know whether *$r* has been aliased to stack or the heap area before that load instruction is executed. For example, an instruction *addi $r6, $sp, 10* will make register *$r6* aliased to stack pointer. Therefore if subsequently a load instruction uses *$r6* as the base pointer, then the compiler needs to ensure that all the locations in the stack be alive at that load instruction. The procedure *compute_register_aliases()* also verifies whether it is possible to perform dead value detection analysis for the current procedure or not. Sometime it may happen that the base pointer aliases could not be determined properly or stack pointer and frame pointer values are changed in the middle of the procedure. In such circumstance dead value detection analysis is not performed on the procedure, since it may produce incorrect information.

After the register aliases are computed, the local and global symbol tables are created. In our compiler analysis, we have maintained the information at the byte level. A variable is created for every memory byte accessed (read or write) by global, stack or frame pointers. A variable is identified by the offset and the base pointer through which it is accessed and these are constant throughout the procedure since these base pointers do not change inside the procedure.

After creating the local and global symbol tables, the *read* and *write sets* [3] are generated for every instruction as shown in Algorithm 1, lines 5 to 32. For load and store instructions read and write sets are created depending on whether they are accessing local or global variables.. We set all the global variables in the *read set* of a call instruction, because a called function can read any global variable. Similarly, if the stack pointer value is passed as a parameter to the called function, all local variables are also set in the *read set* of the call instruction.

For the live variable analysis, we have implemented the algorithm described in [2]. For every basic block, the live variable analysis algorithm computes the set of variables that are live immediately before the basic block and also the variables that are live immediately after the basic block. The only modification done to that algorithm is that we initialize all global variables to be live at the exit of the procedure.

After performing the live variable analysis, the dead value information are computed at the instruction level. and the program is annotated with the dead value information. The

[2]The *control flow graph* is a directed graph representing the structure of the program where the nodes represent the basic blocks and there is a directed edge from basic block B_1 to B_2 if B_2 can immediately follow B_1 in some execution sequence. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

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

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

...			...	
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} :	$sw\ \ r5, 40(GP)$
...			...	
(a)			(b)	

Figure 4: Example code to show the cache optimization using dead value information

pseudo-code for marking dead values are shown in the procedure *mark_dead_values()*. Only marking of the local variables are shown in *mark_dead_values()*. The global variables are processed similarly. A variable x is dead after an instruction *inst* reading x in block B, if *inst* is the last instruction reading 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. Similarly, a variable x is dead after an instruction *inst* writing x in block B, if no instruction after *inst* reads 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.

4.2 Mechanism to Pass the Dead Value Information to the Processor

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 load and store instructions that access that particular memory location. If a load instructions loads from a memory location and after that the 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 compiler changes that store instruction to *store.last*. Consider the example shown in Figure 4. Figure 4(a) shows a sample code. Figure 4(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*.

When the processor executes a *load.last* or a *store.last* instruction it marks the cache location accessed by that instruction as clean. Since the value in that memory location

Algorithm 1 The algorithm for detecting dead values

dead_value_detection (Procedure P)

```
cfg = build_cfg (P);
if (compute_register_aliases (cfg) == FALSE) return;
create_global_symbol_table(P);
create_local_symbol_table(P);
for all instruction instr in P do
  if (instr.opcode == load) then
    if (instr.base_reg == GLOBAL_PTR) then
      var_id = get_var(global_symbol_table, instr);
      set_var (instr.global_read_set, var_id);
    else if ((instr.base_reg == ST_PTR) || (instr.base_reg == FR_PTR)) then
      var_id = get_var(local_symbol_table, instr.);
      set_var (instr.local_read_set, var_id);
    else if (is_aliased(instr.base_reg, GLOBAL_PTR)) then
      set_all_vars (instr.global_read_set, global_symbol_table);
    else if (is_aliased(instr.base_reg, ST_PTR , FR_PTR)) then
      set_all_vars (instr.local_read_set, local_symbol_table);
    end if
  else if (instr.opcode == store) then
    if (instr.base_reg == GLOBAL_PTR) then
      var_id = get_var(global_symbol_table, instr);
      set_var (instr.global_write_set, var_id);
    else if ((instr.base_reg == ST_PTR) || (instr.base_reg == FR_PTR)) then
      var_id =get_var(local_symbol_table, instr);
      set_var (instr.local_write_set, var_id);
    end if
  else if (instr.opcode == function_call) then
    set_all_vars (instr.global_read_set, global_symbol_table);
    if (ST_PTR or FR_PTR passed as an argument to the callee ) then
      set_all_vars (instr.local_read_set, local_symbol_table);
    end if
  end if
end for
live_variable_analysis(cfg);
mark_dead_values(cfg);
```

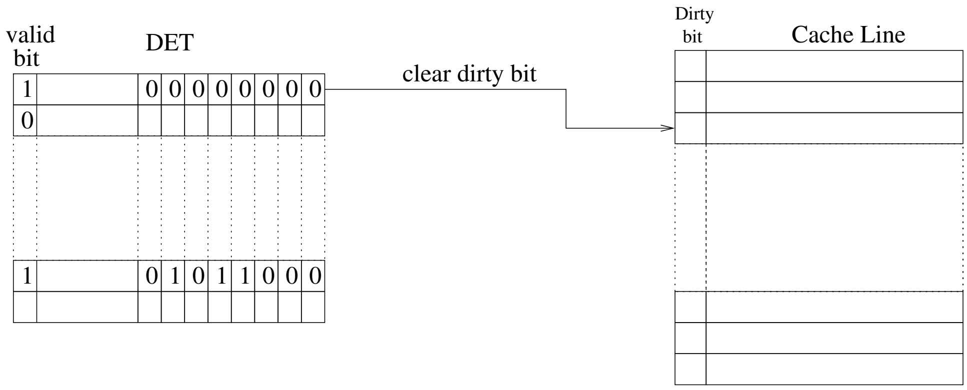


Figure 5: The Structure of DET

would be overwritten anyway before any further use, therefor there is no need to write that value back to memory.

4.3 The Hardware Interface for Dead Value Information

In this subsection we describe the hardware required to process the dead value information obtained from the compiler. Although the compiler, can identify the dead values at the byte level granularity, it passes the dead value information at the word and double-word level in order to reduce the additional hardware required by the processor to maintain that information. Moreover, most of the memory accesses are done at that granularity Therefore, the processor maintains the dirty/clean information of the cache lines at the word level granularity.

So we need one bit for every cache word, i.e. 8 bits in every cache line where cache line size is 32B and word size is 4B. Since this overhead is quite high, we use a separate small table called dead entry table (DET) to store the dirty/clean information of a cache line at the word granularity. 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. The structure of the DET is shown in Figure 5.

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, the value in the flag bits are set(reset) depending on the condition of the cache line during cache write. During cache write, one of the following three scenarios can take place – (i) The write causes a cache miss and a new cache line is allocated and written to and the line is marked dirty - in this case, all the bits in the flag, except for the word(s) where the value is written are set to zero; (ii) The write is a hit in the cache and the line was clean before this write – handled similar to case (i); (iii) The write is a hit and the line was already dirty before this write - all the bits in the flag are set to one, since we do not know which of the word in that cache line is actually dirty.

With subsequent writes the corresponding bits are set. On *lw.last* or *sw.last* if the entry is already there in DET, the corresponding bit is cleared. Also *sw.last* does not set the cache dirty bit. When all the n bits in a DET entry are 0, the processor resets the dirty bit of the cache line. So the 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. If randomly replace an entry.

5 Dead Stack Detection (DSD)

The dead value detection mechanism can be further enhanced by detecting the dead program stack locations in the cache and cleaning those lines on procedure return. In the programs written in languages like C, a procedure allocates a stack in the memory when it is instantiated by decreasing the stack pointer. The stack of the procedure holds the data local to that procedure. The lifetime of the stack is only limited to the lifetime of the procedure. Just before the procedure returns, it deallocates the stack by incrementing the stack pointer. The variables in the stack are dead after the procedure returns.

Therefore any update made to the cache lines corresponding to a procedure's stack need not be propagated to the higher level after that procedure returns since the value in that stack will not be used again. So we can safely mark all the lines in cache 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.

In the following subsections we describe the compiler support and the hardware mechanism proposed by us for the dead stack detection.

5.1 Compiler Support for Dead Stack Detection

Unlike dead value detection, where the compiler has to detect when a memory location is dead, in dead stack detection, the compiler has to specify which stack accesses should outlive the procedure that is accessing it.

At run time, the stack is mostly accessed by using the stack pointer (or frame pointer) [4] as the base pointer. Also, other registers could be used as a base pointer to access a stack location. To make the dead stack detection safe, the hardware and the compiler assume that only the accesses through the stack pointer are in the stack segment. All other accesses are in global[5] segment.

By default, the processor assumes that all the cache lines that are accessed because of a memory access through stack pointer would be dead after the accessing procedure returns. However, sometimes the procedure accesses its callers 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 detect such accesses very easily. 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, this is done along with the DVD phase.

[4] In this work we treat stack and frame pointer in a similar manner. So any reference to stack pointer also includes frame pointer.

[5] We treat heap as a global area

5.2 Hardware Technique 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 since 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 16 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 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. 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.

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 returns, 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.

5.3 Correctness of Dead Stack Detection Mechanism

First of all, our scheme handles the condition that a procedures stack can share a cache line with other procedures' stack or with the global area. The correctness of the above scheme is based on the observation that a dirty line is owned by the procedure that will be returning last. A procedure will never get the ownership of a line marked dirty by its predecessor or a global. If a line has an owner that has higher id value than that of the currently executing procedure, then that procedure has already returned and no longer exists. Because at any point of time, the currently executing procedure is the at the greatest depth among the procedures that are alive at that moment. So it is safe to get the ownership of that line.

Also note that it is not needed to store the lines ownership information at the next level. Consider that procedure A owns a line X and then modifies it. Procedure A calls procedure B and B gets a cache hit for X and modifies the same line X. However the ownership of X still remains with A and the line will be marked clean only when A returns. On the other hand, it might happen that before B accesses X, it gets replaced from the cache. In that case, the modification made by A will be written to the next level. Now when B accesses X, a cache miss will occur and B brings the line again in cache and gets the ownership. Now when B returns, the line is marked clean. This will not cause any error since modifications made by A is already updated in the higher level.

Also note that, when the *call_depth_counter* value exceeds 15, all the cache lines accessed by the procedure stacks, who are at a depth greater than 15, will be cleaned when the procedure at depth 15 returns.

5.4 Example of Dead Stack Detection (DSD)

The key concepts of DSD is explained with the example shown in Figure 6. In the left of Figure 6, a segment of dynamic instruction stream in shown. The program enters procedure *foo()* with the *jsr* instruction. We only show the store instructions in procedure *foo()*. In the right of the Figure 6 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.

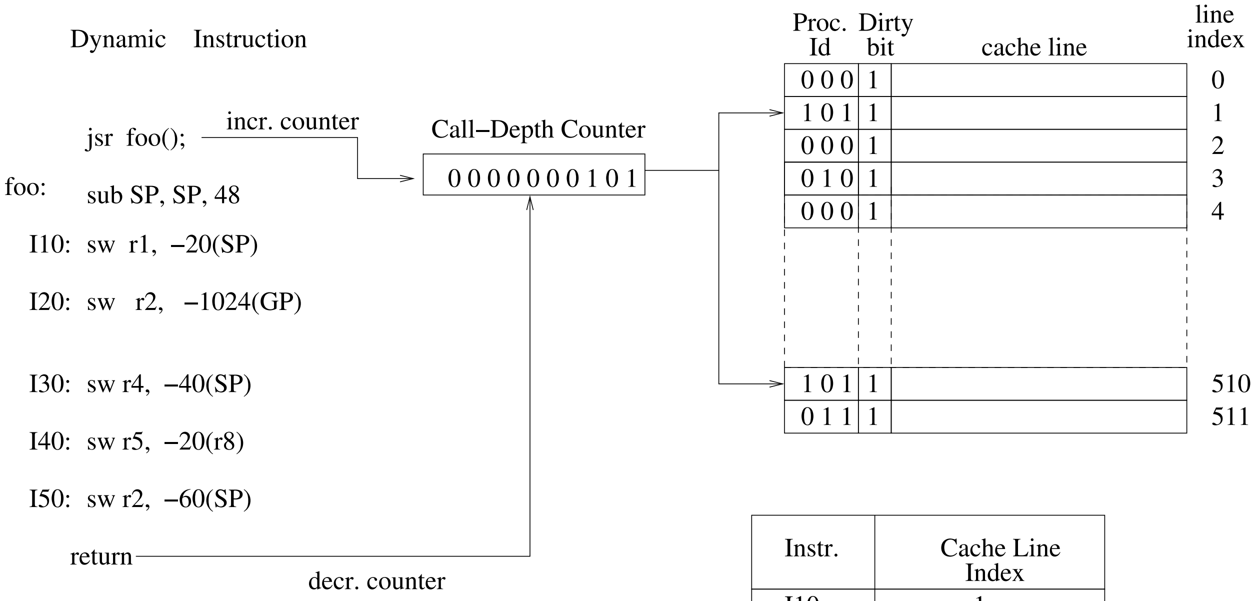
When I10 accesses offset 20 in its activation stack, it is loaded in cache line index 1. Hardware can determine that the access is made to the local stack of *foo()*, because stack pointer SP is used as the base pointer and offset 20 is less than the stack size of *foo()*, which is 48. So the call-depth 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.[6]

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. Since we do not want to discard

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



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

Figure 6: An example showing the cleaning of cache lines accessed by a procedure stack

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.

6 Experimental Evaluation

In this section we discuss some experimental results in lazy-write as well as dead value detection.

6.1 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. to simulate the proposed models. Table 1 shows the parameters of the simulator.

We used the integer and floating point benchmarks from SPEC2000 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 [17]. Table 2 shows the total number of loads and stores committed. We have used a single LWT table for all data caches (L1 and the unified L2). The table is 1K 4-way set associative

Processor Parameter	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 access lat.
L1 - Dcache	32KB, DM, LRU, 32 byte line size,1 cycle access lat.
L2 - Unified	256KB, 4-way set assoc., LRU 64 byte line size, 6 cycle access lat.
LWT	1KB, DM, LRU,
Memory Latency	18 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

cache. The counters are saturating counters of 5 bits each.

6.2 Experiments and Discussion

The first set of experiments show the write back rate at the data. The write back rate (number of write back divided by the number of references), is a good performance metric in here. This is because it is a good measure of the traffic to the memory. As can be seen from Table 3, the AW (Address Wise) and BW (Block Wise) are doing slightly better than the write back policy. This is because the lazy write of a block changes the usage of the block than a conventional write back, and hence the LRU (Least Recently Used) policy is a little bit affected by AW and BW. The block wise is doing a little better than the address wise. This is because is has a little lower prediction rate than the address wise.

The effect of the lazy on LRU is enhancing the performance of the cache in general. For example, Figure 7 shows the miss rates of the unified L2 cache. As it is shown, AW and BW are doing much better than the conventional unified L2. It is to be noted that the IPC (instruction per cycle) is comparable among all the schemes, hence we do not

Integer Benchmark	Write Back .	Address Wise	Block Wise	FP Benchmark	Write Back	Address Wise	Block Wise
bzip2	0.73	0.69	0.63	*ammp*	0.03	0.03	0.02
gcc	6.9	6.9	5.84	*apsi*	2.25	2.23	1.38
gzip	1.1	1.06	0.9	*applu*	1.36	1.35	1.35
mcf	4.86	4.84	4.48	*art*	1.88	1.88	1.11
parser	0.68	0.66	0.53	*equake*	0.32	0.3	0.23
perl	0.17	0.16	0.16	*mesa*	0.66	0.59	0.52
twolf	1.34	1.31	1.21	*swim*	0.04	0.04	0.03
vortex	0.44	0.41	0.35	*wupwise*	0.32	0.31	0.17
vpr	0.97	0.9	0.85				

Table 3: Write Back Rate

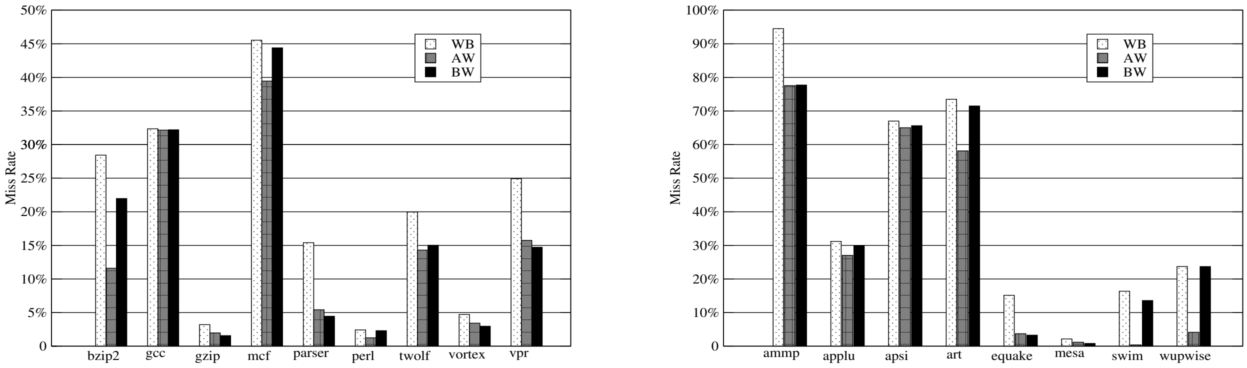


Figure 7: Miss Rate of the Unified L2 Cache

report them here.

6.3 Results of Dead Value Detection

In this section we report the results for DVD and DSD. In Table 4, we report the percentage of times a dirty cache line is cleaned by using either or both of these 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 of DVD and DSD. The results are obtained using a 64 entry DET.

From the table 4 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, then it is not cleaned before caller returns by the time the line may get replaced.

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 4: Statistics for Cleaning Dead lines for DVD and DSD

More detailed register alias analysis could give better improvement for DSD.

DVD is showing a good performance. This implies that the compiler could identify the dead values properly and also this could be effectively used for reducing cache traffic. DVD could identify the dead global variables which DSD cannot. In case of combined scheme, DVD cleans most of the lines before could get a chance to clean them. Therefore we see that in most of the cases in the combined scheme DVD retains its performance.

7 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 the lazy write technique can combine the benefits of both the write through as well as the write back policies. Moreover, it has been shown that the LRU algorithm is positively affected by this method, as indicated by the enhance in unified level 2 cache hit rate. Moreover, compiler techniques were able to clear a lot of dirty blocks without the need to use any bandwidth at all.

The future work includes the study of the effect of the DVD on the traffic by simulating the bus congestion, as well as combining the lazy-write with DVD.

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] Aho, R.Sethi, and J.Ullman. *Compilers: Principles, Techniques and Tools.* Addition-Wesley, 1986.

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

[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] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.