Bandwidth-Friendly Cache Hierarchy

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
Phone:+1-212-650-7310

Anasua Bhowmik
AMD India Engineering Centre
Bangalore, India
anasua.bhowmik@amd.com

Abstract

With the major advances in process technology, several processors, and more sophisticated cache hierarchy, can be embedded on-chip. This opens the door for many interesting and challenging opportunities, by providing high on-chip bandwidth.

However, the off-chip bandwidth becomes a bottleneck, due to the contention on the system bus, the limited number of pins imposed by packaging, and the contention on those pins from the on-chip processors. This negatively affects the whole system performance.

In this paper, we present several techniques to solve the cache-to-memory traffic problem. The first technique, called **bandwidth management***, 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, called* **bandwidth saving***, deals with detecting values that are dead, and hence do not need to be written to the memory altogether, resulting in reduced traffic to the memory, and faster block replacement. The results show that bandwidth management reduced the writeback-rate, and the bandwidth saving cleared more than 30% of the dirty blocks.*

Keywords: cache memory, bandwidth, compiler, prediction

1 Introduction

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 processors and several levels of cache memory on-chip.

However, the cache memory performance is not delivering the required performance. This is due to many reasons, such as 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 lower level, or the main memory. This traffic increases substantially with the increase in the number of processors on chip, as well as with the increasing usage of prefetching techniques [8]. Traffic optimization between cache memories and main memory 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 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 high power consumption as well as severe bus contention. Another important drawback of the write through policy, is that the write operation itself is slow. This mainly is due mainly 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 can get congested because it is not used by the processor alone, some DMA devices, and some devices like the graphics adapter use the system bus, leading to congestion.

Therefore, we need a new policy that can make better use of the 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 *bandwidth management.*
- Using the compiler to find the dead values, and therefore they are not written back to memory, since they will not be referenced again. This technique is optimizing the traffic using *bandwidth*

saving.

The first technique, predicts when a block will not be written again before replacement. Using this information, the block is written back to the memory or 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, detects that a specific value will be dead after a certain instruction, thus, 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. Bandwidth Management is presented in Section 3. The dead value detection techniques are discussed in Sections 4 and 5. Experimental results are shown in Section 6. Finally, Section 7 concludes and summarizes the paper.

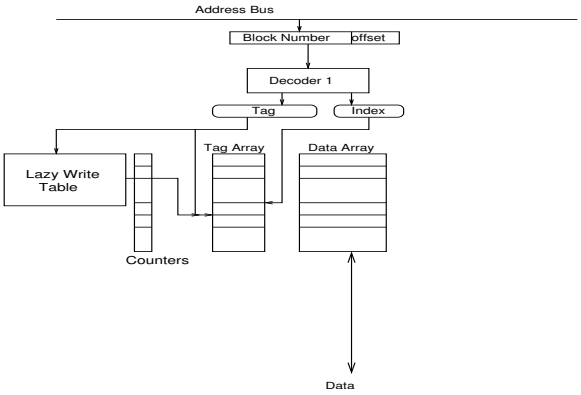

Figure 1: Bandwidth Management Using LWT Table (Address Wise Method)

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 [11] is one of the earliest attempts to do that, followed by many improvements [1][5]. The authors in [13] in proposed a way of using the holes in the direct-mapped cache to decrease conflict misses. In [12], the authors predict a miss and abort the cache access. In [14], 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 [4], by detecting dynamic instruction instances that generate unused results.

A lot of work has been done that focus on compiler analysis and optimization to improve the cache performance [6, 10, 9, 7]. All this work body tries to reduce cache misses by improving data locality. The compiler does so either by placing the data efficiently in memory [6, 10, 9] or by changing the memory access order to improve the temporal and spatial locality.

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 lower level cache, when the bus is idle or an entry is available in the write buffer, if it is dirty, without waiting for the block to be replaced. Therefore, when the time comes for replacing the block, it will be clean and the replacement will be done faster. The whole operation is done outside the critical path, so that 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. We have implemented this idea in two different methods.

The first method depends on a *Lazy Write Table (LWT)*. Figure 1 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 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 LWT entry is updated (i.e. incremented if the new count is bigger than the value stored in LWT or decremented if it is smaller) 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 since it came into the cache. The other counter contains the last number of writes to that block before a replacement. so, the counters are associated with the cache slot, instead of being in a separate table. This has the advantage of adapting to cumulative effect of several applications in case of several processors share the cache. This method works as follows. If the two counters become equal, and the block is dirty, it is written back. When the block is being replaced, the first counter is used to update the second counter (the incremental update similar to the LWT update), and the first counter is reset.

4 Bandwidth Saving: Dead Value Detection (DVD)

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 lower 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.

Sometimes there are multiple *writes* to a memory location without any intervening reads. 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. Similarly, the stack area that has been allocated to a particular dynamic instance of a procedure becomes dead after the procedure returns to the caller.

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.

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 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 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)* 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* [1] 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 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.

5 Bandwidth Saving: 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 proce-

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

Algorithm 1 The algorithm for detecting dead values

dead_value_detection (Procedure P)

```
1: cfg = build_cfg (P);
2: if  (compute_register_aliases (cfg) == FALSE) return;
3: create_global_symbol_table(P);
4: create_local_symbol_table(P);
5: for all instruction instr in P do
6:   if  (instr.opcode == load) then
7:     if  (instr.base_reg == GLOBAL_PTR) then
8:       var_id = get_var(global_symbol_table, instr);
9:       set_var (instr.global_read_set, var_id);
10:    else if  ((instr.base_reg == ST_PTR) || (instr.base_reg == FR_PTR)) then
11:      var_id = get_var(local_symbol_table, instr.);
12:      set_var (instr.local_read_set, var_id);
13:    else if  (is_aliased(instr.base_reg, GLOBAL_PTR)) then
14:      set_all_vars (instr.global_read_set, global_symbol_table);
15:    else if  (is_aliased(instr.base_reg, ST_PTR , FR_PTR)) then
16:      set_all_vars (instr.local_read_set, local_symbol_table);
17:    end if
18:  else if  (instr.opcode == store) then
19:    if  (instr.base_reg == GLOBAL_PTR) then
20:      var_id = get_var(global_symbol_table, instr);
21:      set_var (instr.global_write_set, var_id);
22:    else if  ((instr.base_reg == ST_PTR) || (instr.base_reg == FR_PTR)) then
23:      var_id =get_var(local_symbol_table, instr);
24:      set_var (instr.local_write_set, var_id);
25:    end if
26:  else if  (instr.opcode == function_call) then
27:    set_all_vars (instr.global_read_set, global_symbol_table);
28:    if  (ST_PTR or FR_PTR passed as an argument to the callee ) then
29:      set_all_vars (instr.local_read_set, local_symbol_table);
30:    end if
31:  end if
32: end for
33: live_variable_analysis(cfg);
34: mark_dead_values(cfg);
```

dure 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 lower 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 a 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) [2] 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[3] 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.

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

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

[3] We treat heap as a global area

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.

6 Experimental Evaluation

6.1 Experimental Methodology and Setup

For microarchitectural simulations, we modified the out-of-order processor simulator of the Simplescalar tool set [3], with PISA (portable ISA) instruction set, to simulate the proposed techniques. A post-compilation step is done to insert the generated dead-value information in the binary.

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 [15]. We have used a single LWT table. The table is 1K 4-way set associative cache. The counters are saturating counters of 5 bits each (for the AW method).

Bandwidth management is used in both cache levels (L1 data and shared L2), in order to manage the traffic between caches as well as between the lowest level cache and the main memory. Bandwidth saving techniques are used in the nearest cache to the processor, in order to save the traffic going down the hierarchy.

The rest of the simulator parameters are left with their default values.

6.2 Results of Bandwidth Management

The following 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 an appropriate metric in our case, because it gives an idea about the traffic to the memory. Higher write back rate means more bandwidth requirement.

As can be seen from Table 1, the AW (Address Wise) and BW (Block Wise) are doing better than the write back policy. This is because the early write of a block changes the usage of the block than a conventional write back, and hence the LRU (Least Recently Used) replacement policy is positively affected by AW and BW. BW is doing a little better than the address wise. This is because is has a lower prediction rate than the address wise.

6.3 Results of Dead Value Detection

In this section we report the results for DVD and DSD. In Table 2, 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.

From the table 2 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. 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, 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 proposed techniques on the traffic by simulating system bus congestion.

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] 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.

Table 1: Write Back Rate

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 2: Statistics for Cleaning Dead lines for DVD and DSD

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] T. Chilimbi, B. Davidson, , and J. Larus. Cache conscious structure definition. In *PLDI*, 1999.

[10] T. Chilimbi, M. Hill, , and J.Larus. Cache conscious structure layout. In *PLDI*, 1999.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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