Locality vs. Criticality *

Srikanth T. Srinivasan+ Roy Dz-ching Ju‡ Alvin R. Lebeck+ Chris Wilkerson‡

+Department of Computer Science
Duke University
{sri, alvy}@cs.duke.edu

‡Microprocessor Research Labs
Intel Corporation
{roy.ju, chris.wilkerson}@intel.com

Abstract

Current memory hierarchies exploit locality of references to reduce load latency and thereby improve processor performance. Locality based schemes aim at reducing the number of cache misses and tend to ignore the nature of misses. This leads to a potential mis-match between load latency requirements and latencies realized using a traditional memory system. To bridge this gap, we partition loads as critical and non-critical. A load that needs to complete early to prevent processor stalls is classified as critical, while a load that can tolerate a long latency is considered non-critical.

In this paper, we investigate if it is worth violating locality to exploit information on criticality to improve processor performance. We present a dynamic critical load classification scheme and show that 40% performance improvements are possible on average, if all critical loads are guaranteed to hit in the L1 cache. We then compare the two properties, locality and criticality, in the context of several cache organization and prefetching schemes. We find that the working set of critical loads is large, and hence practical cache organization schemes based on criticality are unable to reduce the critical load miss ratios enough to produce performance gains. Although criticality-based prefetching can help for some resource constrained programs, its benefit over locality-based prefetching is small and may not be worth the added complexity.

1 Introduction

The wide-spread use of cache memory hierarchies [17] is a testament to their success as a cost-effective solution to help alleviate the growing disparity between processor cycle time and DRAM access time. By using a small, fast, but expensive SRAM cache to satisfy most accesses, cache hierarchies can provide access times close to processor speeds, while cheaper, slower DRAM provides large capacity to avoid I/O delays. Increases in transistor budgets enable most systems today to employ two or three levels in a cache hierarchy, with increasing capacity and access time as you move from the processor to main memory.

Caches work by exploiting spatial and temporal locality in a program's access pattern. Spatial locality is exploited by fetching a region of memory—called a cache block—rather than just the accessed data. Caches exploit temporal locality by retaining recently accessed cache blocks. Most cache management schemes try to exploit locality to increase the fraction of memory accesses satisfied by the cache (i.e., cache hit ratio).

Although increasing the overall number of cache hits is usually desirable, recent research [19] shows that not all memory accesses are equal. In dynamically scheduled processors, the latency of some memory load operations can have a much larger influence on overall performance than other loads. Therefore, it may be possible to improve overall performance by decreasing the latency of these critical loads at the expense of increased latency for non-critical loads.

The goal of this paper is to determine if criticality is a strong enough program property to warrant a change in memory hierarchy management techniques for practical implementations. Specifically, we investigate if practical criticality-based approaches can equal or surpass the performance of existing locality-based techniques. The previous analysis [19] relied on a sophisticated simulator with roll-back to determine load criticality. A practical implementation requires hardware support for on-line computation of load criticality. This must be augmented by realistic load latency reduction techniques that can exploit this information.

In this paper, we propose a hardware implementation to determine load criticality and validate that there is potential to improve performance over a traditional cache hierarchy. We then compare locality-based cache organization and prefetching techniques with corresponding criticality-based schemes. Our criticality based cache organization scheme uses a critical cache, that is functionally similar to a conventional victim cache [10], but holds only blocks that were touched by a critical load. We also examine multi-line

* Appears in Proceedings of the 28th International Symposium on Computer Architecture, June 2001.

prefetching [9, 12] based on both locality and criticality.

We use Simplescalar [1] to evaluate the above techniques for a set of SPEC 2000 integer and Olden [16] benchmarks. The primary result from our simulations is that load criticality is not a sufficiently strong property to warrant cache management changes. Although, criticality can be used as a filter for making prefetching decisions, it does not appear to be worth the added complexity. In particular, our results reveal the following:

- There is potential to exploit criticality. If all the hardware identified critical loads could be satisfied by the L1 cache, it would produce an average 40% improvement over a traditional memory hierarchy. This compares to an average 12% improvement if an equal percentage of loads are randomly chosen to hit in the L1 cache.
- Managing cache content based on locality outperforms criticality-based techniques. The working set of critical loads is large and because of spatial locality between non-critical and critical loads, a locality-based cache provides lower critical load miss ratios than a criticality-based cache.
- Criticality-based prefetching into the L2 cache achieves an average speedup of 4%, compared to 2% for locality-based prefetching, across several benchmarks with resource constraint problems. However, if resource constraints are not a problem, the most aggressive locality-based prefetching scheme does best.

The remainder of this paper is organized as follows. Section 2 provides background on load criticality, presents our hardware technique for determining load criticality and evaluates the potential for improved performance. We evaluate cache organization schemes in Section 3 and Section 4 investigates prefetching. Section 5 discusses related work and Section 6 concludes this paper.

2 Load Criticality

Our previous work shows that there is variation in the latency requirements for individual loads and that conventional cache hierarchies may not always match the latency demands of a load [19]. Based on their latency requirements, loads are classified as critical and non-critical. Loads that must complete early to avoid performance degradation are critical and those that can tolerate long latencies are non-critical. To eliminate the discrepancy between latency demands and actual incurred latency, we must 1) identify the critical loads and 2) provide a memory hierarchy that satisfies critical loads with minimal latency. The rest of this section focuses on identifying critical loads. We investigate memory hierarchy design in Sections 3 and 4.

2.1 Online Critical Load Classification

Our previous work on load latency uses sophisticated simulation with rollback to determine how long a load could remain outstanding without degrading performance relative to an ideal memory system where all loads hit in the L1 cache. While this is a reasonable approach for limit studies, it is unacceptable for practical use. However, many of the insights from that study can be used to develop a practical implementation for determining load criticality.

The previous study shows that load criticality is determined, to a large extent, by the chain of instructions dependent on the load. In particular, the type of dependent instructions (e.g., mispredicted branch) indicates if the load is likely to degrade processor performance. Similarly, the number of instructions in its dependence chain indicates if a long latency load will stress the processor's finite resources. If most instructions issued after a load are dependent on the load, the processor may stall, unable to find independent instructions to execute in subsequent cycles.

From these observations, we can construct hardware to determine load criticality by monitoring the type of instructions in a load's dependence chain, and counting the number of independent instructions issued in cycles immediately after the load. In our design, a load is classified as critical if it satisfies any of the following criteria: 1) The load feeds into a mispredicted branch, 2) The load feeds into another load that incurs an L1 cache miss, or 3) The number of independent instructions issued in an N cycle window following the load is below a threshold.

Our hardware design is based around the Register Update Unit (RUU), Load/Store Queue (LSQ) model for out-of-order instruction scheduling and speculative execution [18]. We simulate a processor with the five pipeline stages: 1) Fetch, 2) Decode/Register Rename, 3) Issue ready instructions to functional units, 4) Writeback (supply results to dependent instructions), and 5) Commit results to the register file in program order.

2.2 An Implementation

To track a load's dependence chain, we add a hardware structure, called the Criticality Table (CT). Each RUU entry has a corresponding CT entry with a cache flag, a branch flag, and a dependence vector. The cache flag is used only for load instructions, and indicates if the load missed in the L1 cache or not. The branch flag is used for branch instructions and a set flag indicates that the branch was mispredicted. The dependence bit-vector has one bit per LSQ entry. If bit j of the dependence vector is set for RUU entry i, then instruction i is dependent on the j^{th} entry in the LSQ. The LSQ holds both loads and stores, but since only loads have instructions dependent on them, bits that are set

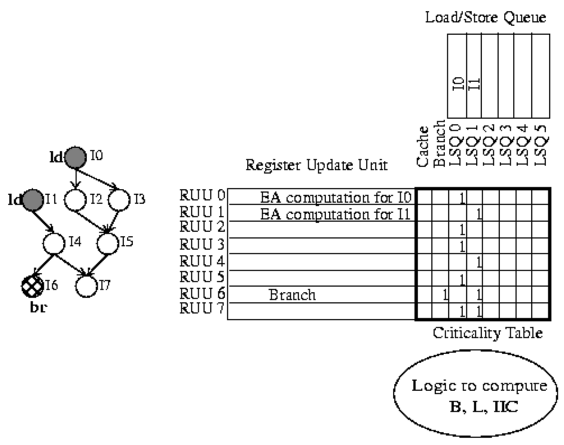


Figure 1. Criticality Table

in the dependence vectors indicate load dependencies alone.

The CT is updated at different stages of the processor pipeline. In the Decode stage, the dependence check logic is extended to set an instruction's dependence vector bits in the CT. The branch flag in the CT entry can be set after the corresponding branch is resolved, in the Writeback stage. Similarly, as soon as the L1 or L2 cache access for a load is completed, the cache flag of the load's CT entry can be updated using the hit/miss information.

Consider the Criticality Table and example dependence graph shown in Figure 1. The operands of I7 are produced by instructions I4 and I5. I4 depends on load I1, while I5 depends on load I0. From the dependence graph, it is clear that I7 depends on both loads, I0 and I1. To set the dependence vector of I7, the dependence information of instructions I4 and I5 are propagated down to I7. We modify the dependence check logic to set the dependence vector for I7 by performing a bitwise OR of the dependence vectors of I4 and I5. At any instance of time, dependence bits along a row in the CT indicate which currently active loads a particular instruction depends on, while the bits along a column correspond to the instructions dependent on a specific load. The number of instructions dependent on a load is given by the sum of the column in the CT corresponding to the load. The column sum of LSQ entry 0, that corresponds to load I0, is five, which is the same as the number of instructions dependent on I0 (including itself) in the dependence graph.

To determine if a mis-predicted branch is dependent on a load (B) we compute a bit-wise AND of the column in the CT corresponding to the load with the branch flag column. If the result is non-zero, then the load has a mis-predicted branch dependent on it. Similarly, to determine if a load feeds into another load that misses in the cache or not (L) we perform a bitwise AND of the cache flag column with the load's column in the CT.

We also associate a counter called the Independent Issue Counter (IIC) with each load. We monitor all instructions issued in a window of N cycles following the load's issue. For each instruction issued during that window, if it is independent of a load, we increment the load's IIC. At the end of N cycles, if a load's IIC is less than an issue threshold, the load is classified as critical. In this paper, we present results with a window of 8 cycles (N = 8), and an issue threshold of 16. We find that using different values for N and the issue threshold do not change the nature of our results.

The above scheme can determine the load criticality only after a window of N cycles after a load issues. However, many schemes that exploit criticality require the classification information before a load is issued. For this purpose, we use a load criticality predictor similar to a two level branch predictor. It uses two bits of global branch history and a 4096 entry load criticality history table, and achieves load criticality prediction rates ranging from 73% to 95% across our benchmarks. Using global branch history serves to include path information leading to a load in the criticality predictor. We find that having more global branch history bits improves the prediction accuracies only slightly.

On average, about 30% of all dynamic loads are classified as critical, 12% based on the Branch criteria, 5% based on the Cache criteria and 13% based on the Issue criteria. The remainder of this section shows that if these critical loads are captured in the L1 or L2 cache, there is substantial opportunity for performance improvement.

2.3 Potential Performance Improvement

We use Simplescalar [1] to evaluate the potential benefits of our critical load classification scheme on a subset of the SPEC2000 integer benchmarks and two pointer-intensive benchmarks from the Olden suite [16]. We fast forward the first 4 billion instructions, using them to warm-up the caches, and gather results from detailed execution of the next 100 million instructions.

Our baseline processor can issue 8 instructions per cycle and has 256 RUU entries and 128 LSQ entries. We simulate a traditional memory system consisting of a 8KB, 2-way set associative L1 cache with 16B lines and a one cycle latency. The L2 cache is 256KB, 2-way set associative with 64B lines. The L2 access latency is 5 cycles and the transfer time on the bus is 4 cycles. All caches have 128 MSHR entries [11] and use an LRU replacement policy. We use an infinite main memory with a request time of 2 cycles, a latency of 238 cycles and a 60 cycle transfer time. These parameters are used throughout the paper, unless otherwise stated. We note that experiments with a 32KB L1 cache with 3 cycle latency, a 1MB L2 cache with a 20 cycle latency, and two different memory latencies of 60 cycles and 300 cycles, lead to similar conclusions.

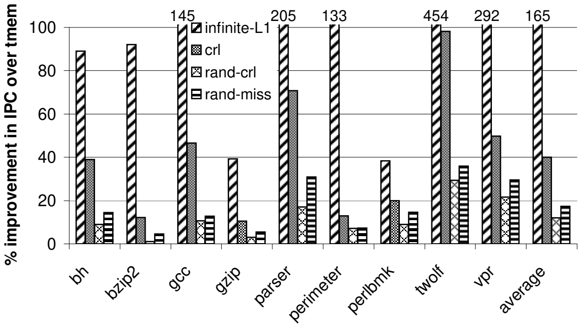


Figure 2. Potential Criticality-based Performance Improvement at L1 cache level

We compare the performance of a traditional memory system (*tmem*) with that of a memory system in which all predicted critical loads are guaranteed to hit in the cache (*crl*). For the crl runs, we modify our simulator such that if a predicted critical load misses in the cache, we force a cache hit by choosing a victim block and updating the tags and data instantaneously with the critical load's tag and data. We also compare the crl scheme to two other schemes, rand-crl and rand-miss. The rand-crl scheme randomly classifies a load as critical such that the percentage of loads classified as critical matches that of the crl scheme and forces them to hit in the cache. The rand-miss scheme randomly converts the same proportion of load misses to hits as the crl scheme.

Figure 2 shows the percentage improvement in Instructions Per Cycle (IPC) over a traditional memory system when loads are forced to hit in the L1 cache. The case where all loads hit in the cache (shown as infinite L1 in Figure 2) represents an upper bound on the performance improvements possible. Using the crl scheme we can achieve 40% improvement in performance, on average, if all critical loads hit in the L1 cache, whereas the average performance benefits for rand-crl and rand-miss are only 12% and 17%, respectively.

The average performance improvements at the L2 cache level are 43%, 20%, and 31% for the crl, rand-crl, and rand-miss schemes, respectively. These results suggest that our classification scheme does a good job of identifying the set of performance critical loads. We note that, in general, forcing loads to hit in the L2 cache achieves higher performance than forcing loads to hit the L1 cache. This is because the L2 block-size is larger than the L1 block-size, and forcing loads to hit in the L2 cache creates additional L2 hits.

If loads that are classified as critical based on the Branch criteria alone are forced to hit in the L1 cache, we see average performance gains of 9%. Similarly, we see average improvements of 15% for the Cache criteria, and 16% for the Issue criteria. This suggests that using the individual criteria alone for classifying loads as critical will drastically reduce the potential performance improvements. Therefore, in this paper we use all three criteria for classifying loads as critical.

The results presented in this section indicate there is potential to exploit information on load criticality to significantly improve performance. The challenge is to deliver this potential with realistic implementations. The proposed hardware technique for determining load criticality is just one part of the solution. The following sections investigate the remaining parts: criticality-based cache management and prefetching policies.

3 Cache Organization

The most significant component of a load's latency is the level in the memory hierarchy where the accessed data resides. Performance may improve if critical loads can be satisfied by caches close to the processor even if non-critical loads suffer increased misses. This section investigates criticality-based cache organization.

The central idea behind criticality-based caching is to keep critical data[1] close to the processor at the potential expense of non-critical data. To achieve this, we investigate both a new cache replacement policy and a different cache organization. In this paper, we focus on an alternative cache organization called a critical cache. However, as we discuss later in this section, our results are similar for modified replacement techniques.

A critical cache serves as a victim cache [10] for critical data. When a primary cache sub-block, equal to the critical cache block size, is touched by a critical load, we set a corresponding critical bit. When the block is replaced from the primary cache, the sub-blocks with the critical bit set are copied to the critical cache. Our design uses a smaller block size for the critical cache than the primary cache, to be able to store only the critical sections of a cache block.

On every reference, both the primary cache and the critical cache are accessed in parallel. If the requested data is found in either of the cache structures, it is treated like a cache hit. When a reference hits in the critical cache, all sub-blocks that are part of the primary cache block are transferred to the primary cache and, if necessary, a request is sent to the next level in the memory hierarchy to fetch any missing sub-blocks.

The equivalent locality based caching scheme consists of a locality cache alongside the primary cache. The capacity, block-size and associativity of the locality cache match those of the critical cache. Any block that is evicted from the primary cache is transferred to the locality cache which helps retain both critical and non-critical data longer than

[1] We define critical data as data touched at least once by a critical load while resident in the cache.

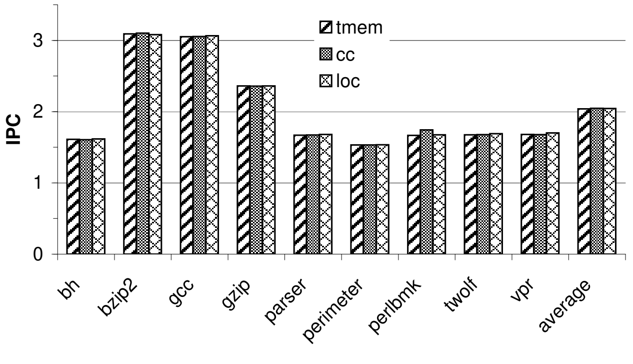


Figure 3. L1 Content Management: IPC

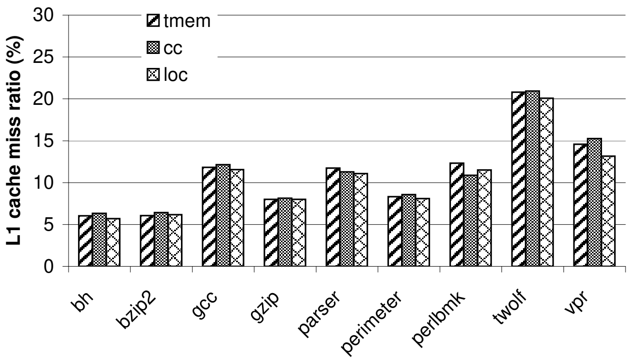
just the primary cache. The function of our locality cache is similar to a conventional fully-associative victim cache. However, we want a sufficiently large capacity, and hence use lower associativity to keep access time low. The remainder of this section compares a critical cache with an equal sized locality cache at both the L1 and L2 cache levels.

3.1 Locality vs. Criticality at the L1 Cache Level

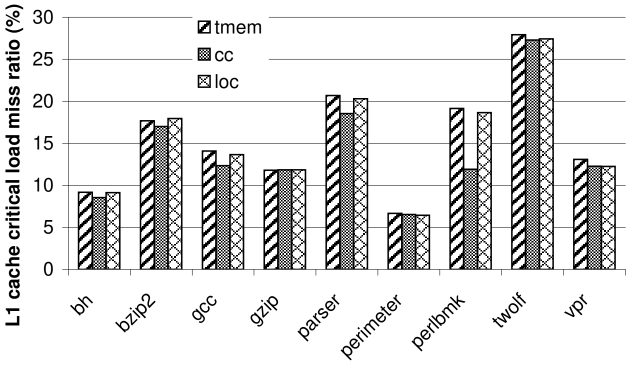
For experiments at the L1 cache level, we model an infinite L2 cache to eliminate the L2 cache performance effects. Our critical/locality cache configuration consists of a 4KB 2-way set associative L1 cache with 16B lines and a 4KB 2-way set associative critical/locality cache with 8B lines. Even though the sum of the cache sizes in the critical cache configuration is equal to the primary cache size in a traditional memory system (tmem), it is different from splitting tmem into a critical half and a non-critical half, because of the smaller block-size of the critical cache. Also, in general, the critical/locality cache could have a higher associativity than the primary cache. However, simulations of up to 8-way associative critical/locality caches reveal no significant changes in our results.

Figure 3 compares the IPC numbers of a critical cache (cc) and a locality cache (loc) to tmem. These results show there is little change in the IPC numbers compared to tmem for most benchmarks. The exception is perlbmk, where a critical cache improves performance by 5%.

Figure 4 shows the overall L1 cache miss ratio and the critical load miss ratio for the three configurations. The critical cache achieves the lowest critical load miss ratios among the three configurations, in-line with our expectations. The critical cache however, increases the overall miss ratio for all benchmarks except parser and perlbmk, while the locality cache decreases the overall miss ratio for all benchmarks. The reductions in the critical load miss ratios compared to tmem are not significant enough to translate into noticeable performance gains, except for perlbmk.



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 4. L1 Content Management: Miss Ratio

3.2 Locality vs. Criticality at the L2 cache Level

To evaluate criticality-based caching at the L2 cache level, we use two different configurations. One configuration has a 128KB 2-way set associative critical cache with 16B lines alongside a 128KB 2-way set associative L2 cache with 64B lines. The second configuration has an 8KB 2-way set associative critical cache with 16B lines along with a 256KB 2-way set associative L2 cache with 64B lines. For comparison, in each configuration the size of the locality cache matches the critical cache size.

Figure 5 shows the IPC values for each L2 cache organization. From this data we see that neither critical cache configuration achieves performance benefits over a traditional memory system. Moreover, when half of the available cache space is used as a critical cache (cc:128k+128k), performance drops by 21% for bzip2 and by 31% for gzip. gzip with the loc:128k+128k configuration, is the only case that achieves noticeable performance improvement (8%) over a traditional cache.

By examining the overall miss ratio (see Figure 6a) we see that cc:128k+128k increases the overall miss-ratio for all benchmarks except bh, twolf and vpr. We also observe

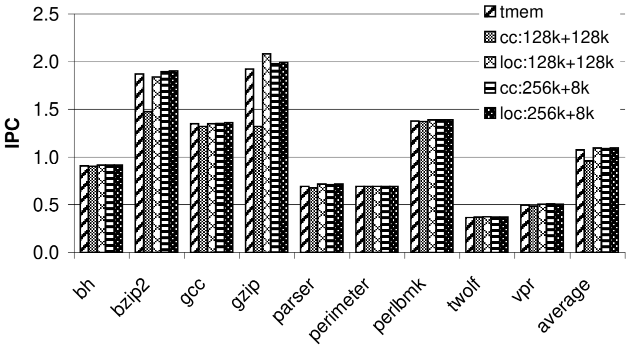
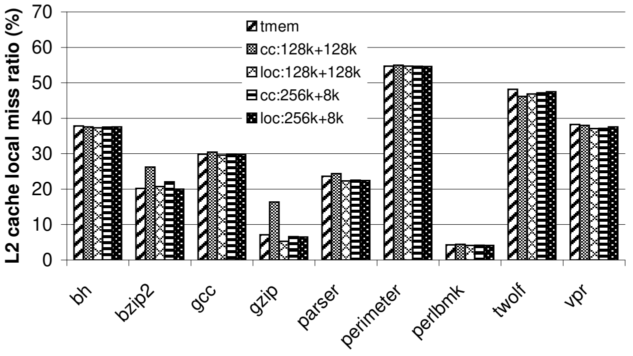
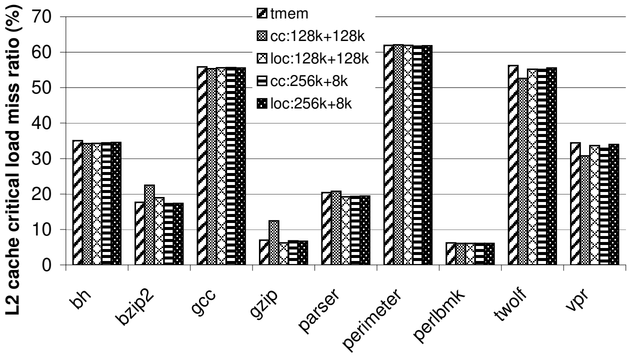


Figure 5. L2 Content Management: IPC



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 6. L2 Content Management: Miss Ratio

that the locality-based cache performs comparably to the criticality-based scheme for most benchmarks. From the critical load miss ratios, shown in Figure 6b, we see there is no significant reduction in the critical load miss ratio for the critical cache or locality cache configurations. Furthermore, in contrast to our expectations, bzip2 shows an increase in the critical load miss ratio for both the 128k+128k locality/critical cache configurations, while gzip suffers an increase in the critical load miss ratio for the 128k+128k critical cache configuration. These increased miss ratios correlate with a commensurate decrease in performance relative to a traditional memory system.

The above results indicate that retaining critical blocks longer by means of a critical cache does not decrease the critical load miss ratio enough to improve performance. We arrive at similar conclusions from experiments with a criticality-based cache replacement scheme that uses an aging policy to let critical blocks remain in the cache longer, and bypasses non-critical data when necessary. The premise of criticality-based cache organizations or replacement policies is that critical loads are a fraction of all references, and hence their working set will be much smaller than the overall program working set. The smaller working set should reduce competition in the criticality-based caches and produce lower critical load miss ratios than traditional caches based on locality. However, our simulations do not show significant reductions in critical load miss ratios for criticality-based caches.

3.3 Critical Load Working Set

One possible explanation for the above results is that the programs access too much critical data. If this is true, it will be difficult for any practical cache organization or replacement policy to exploit criticality information. To further explore this hypothesis and to gain insight into the above results, the remainder of this section examines the working set sizes of our benchmarks.

We measure working set size by determining the smallest cache size required to obtain a specific miss ratio. We simulate cache sizes from 4KB to 16MB, with an associativity of 8, to minimize conflict misses, and a fixed cache line size of 16B. Since we are only interested in cache miss ratios and since the number of simulations required is large, we use trace driven simulation instead of detailed performance simulation. Using Simplescalar, we collect a reference trace for each of the benchmarks that includes criticality information for loads. We skip the first 3.9B instructions and gather traces over the execution of the next 200M instructions. In our cache simulations, the first 100M of the trace instructions are used to warm-up the caches. Thus, the period of execution we evaluate using traces matches the simulations in the rest of the paper.

To estimate the overall working set, we simulate traditional caches managed based on locality and obtain the overall miss ratios (locality.all). To determine the critical load working set, we simulate criticality-based caches that allocate cache blocks only on critical load misses and obtain the critical load miss ratios (criticality.crl). We also note the critical load miss ratios for the locality based runs (locality.crl) which indicates how well the locality based caches

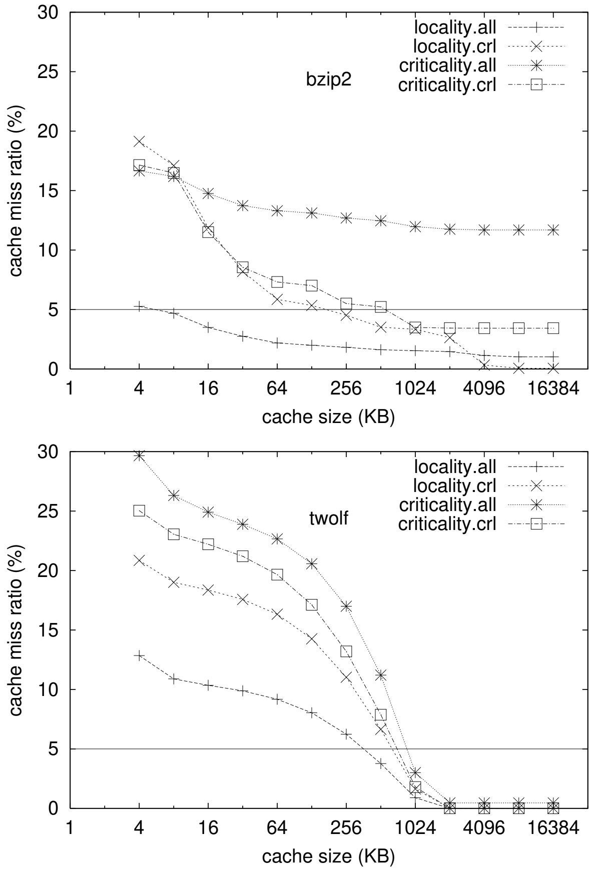


Figure 7. Working Set

are able to capture the critical load working set. For the sake of completeness, we also present the overall miss ratios for the criticality based runs (criticality.all). Figure 7 plots the overall and critical load miss ratios for both the locality and criticality based cache configurations. We provide data for only two benchmarks due to space limitations, however the other benchmark results are qualitatively similar.

Assume we define the working set as the smallest cache size required to capture 95% of the references (a 5% miss ratio, the solid line in Figure 7). To obtain the *overall working set*, we find the smallest cache size that captures 95% of *all references* (*overall miss ratio* less than 5%), whereas to get the *critical load working set*, we determine the smallest cache size that captures 95% of the *critical load references*.

Table 1 shows the overall and critical load working sets for our benchmarks. From Table 1, we see most benchmarks have quite large critical load working sets, 16KB to 4MB. The benchmark perlbmk, has the smallest critical load working set size of 16KB and it is also the only benchmark whose critical load working set is comparable in size

Benchmark	Overall Working Set	Critical Load Working Set	
	locality	locality	criticality
bh	4	128	128
bzip2	8	256	1024
gcc	32	4096	4096
gzip	4	128	128
parser	32	512	1024
perlbmk	16	16	16
twolf	512	1024	1024
vpr	64	1024	1024

Table 1. Working Set: Cache size in KB required to achieve 95% cache hit ratio.

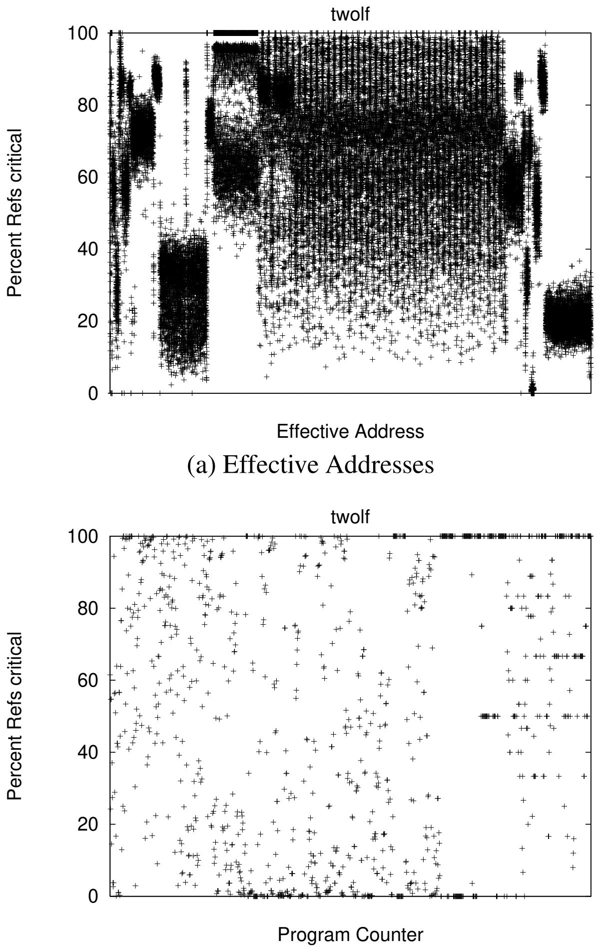
to the overall working set. For these reasons, perlbmk is the only benchmark for which the critical cache shows performance gains at the L1 cache level, as seen earlier. Note that perlbmk's working set size is much smaller than our L2 cache size and hence a critical cache next to the L2 cache is not of much use.

The large critical load working sets are due to the fact that critical references are not concentrated on a few effective addresses, but rather are spread across a large portion of the virtual address space. This can be clearly seen from Figure 8(a) that shows the percentage of references that are critical, for each effective address (32B block) touched by twolf. This is true for the other benchmarks also.

This distribution of critical references across most of the effective addresses can in turn be attributed to different load PCs that touch different effective addresses becoming critical over the lifetime of a program. Figure 8(b) shows a typical criticality distribution per PC. From Figure 8(b), we see that the criticality of load PCs vary, and that most load PCs are classified as critical at some point during program execution.

The instability in the criticality of load PCs can be understood by examining the individual critical load classification criteria. The Branch criteria classifies a load as critical based on whether a dependent branch is mispredicted or not. Similarly, a load is classified as critical by the Cache criteria if a dependent load misses in the L1 cache or not. The number of independent instructions for a load (needed for the Issue criteria) is often determined by whether branches in the vicinity of the load are taken or not. The overall criticality distribution of load PCs reflects a combination of the distributions of branch mis-predictions, L1 cache misses, and branch directions.

Thus the very nature of programs and their execution, leads to large working sets for critical loads, and prevents criticality-based content management schemes from reduc-



(a) Effective Addresses

(b) Program Counters

Figure 8. Critical Load Distribution

ing competition by reducing the working set. This is the primary reason why criticality-based cache organizations do not decrease the critical load miss ratios significantly compared to a traditional locality-based memory system.

From Figure 7 we also observe that in most cases, locality-based caches achieve lower critical load miss ratios than criticality-based caches. This is because in locality-based caches, non-critical misses can prefetch some critical data due to spatial locality between critical and non-critical sub-blocks within a block. Any criticality-based scheme that prevents non-critical misses from allocating a cache block, discounts the possibility of future critical accesses to the entire cache block and hence fails to exploit this property. This leads to a higher number of compulsory critical load misses for criticality-based caches than locality-based caches. This effect can be observed in the case of bzip2 in Figure 7. For bzip2, the critical load miss ratio settles down at 3.5% for criticality-based caches beyond a size of 1MB, while it continues to drop to almost 0% for locality-based caches. This places the criticality-based cache organization schemes at a further disadvantage compared to locality-based schemes.

3.4 Discussion

The results in this section suggest that locality is a better property than criticality for managing cache content. The goal of the criticality-based schemes discussed so far has been to retain critical data in the cache longer and thereby convert subsequent critical load cache misses into hits. However, since the critical load working set is very large, simply modifying cache replacement or allocation policies alone is not sufficient to retain critical data for long enough periods of time to achieve significant reductions in the critical load miss ratio.

To move toward the 40% performance improvements that could be obtained by exploiting information on critical loads, we must investigate alternate solutions. Prefetching is a technique that can be used to bring in critical data just ahead of use. The advantage of prefetching over caching is that it does not require cache space to hold critical data for extended periods of time. We study prefetching for critical loads in the next section.

4 Criticality-Based Prefetching

Prefetching exploits the spatial nature of accesses and the regularity of access patterns by predicting future accesses and initiating a memory access even before a load is actually issued. Accurate address prediction, proper timing, minimal cache pollution and minimal interference with regular load requests are essential ingredients of a good prefetching technique.

Early research concentrated on prefetching for all loads [3, 6, 9, 12, 13]. Our goal is to reduce cache pollution and resource requirements by prefetching for critical loads alone. These selective prefetching techniques are aimed at achieving improvements in critical load cache hit ratios, thereby leading to higher performance.

We evaluate two proposed prefetching schemes: the Reference Prediction Table (RPT) [3] and the Spatial Footprint Predictor (SFP) [12]. RPT keeps track of the stride (current effective address - previous effective address) of each load PC, and issues a prefetch whenever a steady stride is observed. SFP groups multiple cache blocks into macro-blocks, and records the blocks within each macro-block that are touched—called the spatial footprint—during the macro-block's lifetime in the cache. When a miss to one of the blocks occurs, the entire spatial footprint of the macro-block is prefetched.

Comparing the two prefetching schemes RPT and SFP, our simulations show that SFP produces superior performance improvements for most of the benchmarks. Hence, we choose SFP over RPT for further evaluation. We make SFP critical load centric (sfp-crl) by modifying it to prefetch only the spatial footprint of critical loads, and refer to the original SFP scheme as sfp-all.

We also evaluate another critical load centric prefetching scheme, called mbp-crl (Multiple Block Prefetch on critical load miss). mbp-crl prefetches an entire macro-block on a critical load cache miss and reverts to regular cache line fetch if a cache miss is from a non-critical load. The motivation for mbp-crl is based on the following observation from our criticality analysis experiments: Cache lines that are nominated by a critical load (i.e. brought into the cache due to a critical load miss) are more likely to be referenced by subsequent critical loads than cache lines that are nominated by a non-critical load. We find that the average number of additional words touched by critical loads within a cache line is 4-5 times higher for cache lines nominated by a critical load than for cache lines nominated by a non-critical load. mbp-crl exploits this property at the macro-block granularity by prefetching an entire macro-block, whenever a cache line is nominated by a critical load, and thereby hopes to achieve more critical load hits.

The locality based scheme corresponding to mbp-crl is mbp-all, which prefetches an entire macro-block on all load misses. Store/write misses still fetch only one cache line which allows for finer granularity replacements, and thus mbp-all is different from having bigger cache lines.

To summarize the different prefetching schemes, consider a sample macro-block that consists of 4 cache blocks. Assume that blocks 0, 1, and 3 constitute the spatial footprint of the macro-block, while the spatial footprint of critical loads consists of blocks 1 and 3. If there is a load miss to any of the blocks in the macro-block, sfp-all will prefetch blocks 0, 1, and 3, sfp-crl will prefetch blocks 1 and 3, and mbp-all will prefetch the entire macro-block. mbp-crl will prefetch the entire macro-block only if there is a critical load miss to blocks 1 or 3. In the remainder of this section, we compare two criticality based prefetching schemes (sfp-crl and mbp-crl) with their corresponding locality based schemes (sfp-all and mbp-all) at both the L1 and L2 cache levels.

4.1 Prefetching Into the L1 Cache

When prefetching into the L1 cache, we simulate an infinite L2 cache to isolate the effects of prefetching into the L1 and L2 caches. We use an L1 cache line size of 16B and the macro-block size of 64B. We initiate prefetches on an L1 cache miss and check the L1 cache tags to issue prefetches only for those blocks that are not already in the cache. We find through experiments that queuing regular load requests the earliest in the MSHRs, followed by prefetch requests, followed by store misses, performs best, and hence, we use this ordering for all our runs.

From our simulations, we find that at the L1 cache level, none of the prefetching schemes improve performance. In fact, many decrease performance. The main reason for the performance degradation is the resource constraints introduced by the prefetch requests. First, they occupy MSHR entries and deny MSHR entries for regular load requests, thereby delaying regular loads. Second, once a prefetch request has acquired the L1–L2 bus, a regular load request has to wait for the data transfer to be complete before acquiring the bus. This adds additional delays for regular loads.

From our experiments, we also observe that for the benchmarks bzip2, parser, and vpr, the MBP schemes increase either the overall miss ratio or the critical load miss ratio, suggesting that pollution also becomes a problem at the L1 cache level. For the rest of the benchmarks, all prefetching schemes decrease both the overall and critical load miss ratios. However, the increase in the resource constraints due to prefetching offsets the advantages due to lower miss ratios. This results in either meager performance improvements of up to 2% or decreases in performance of up to 43% and 14% for the locality and criticality based schemes respectively, compared to a traditional memory system. Hence, we do not look any further at locality/criticality based prefetching into the L1 cache, and instead, turn to the L2 cache.

The results at the L1 cache level might not hold at the L2 cache level for two reasons. First, L2 caches are usually much bigger than L1 caches and so the chances of pollution in the L2 cache are lower than in the L1 cache. Secondly, the strain on the MSHR queue and the bus to the next level of memory could be very different due to the difference in the density/frequency of accesses to the two levels.

4.2 Prefetching Into the L2 Cache

When prefetching into the L2 cache, we use an L2 cache line size of 64B and a macro-block size of 256B. We make a prefetching decision (whether to prefetch or not and what to prefetch) whenever there is an L2 cache miss. Figure 9 shows the percentage improvement in IPC of the four prefetching schemes over a traditional memory system. Unlike at the L1 cache level, prefetching into the L2 cache shows performance gains in most cases. vpr is the only benchmark that does not show any performance benefits.

In general, whenever SFP increases performance, the locality based sfp-all produces higher performance numbers than the criticality based sfp-crl. This is not always true for MBP. For gcc, gzip, and perimeter, mbp-all does produce considerably higher performance improvements than mbp-

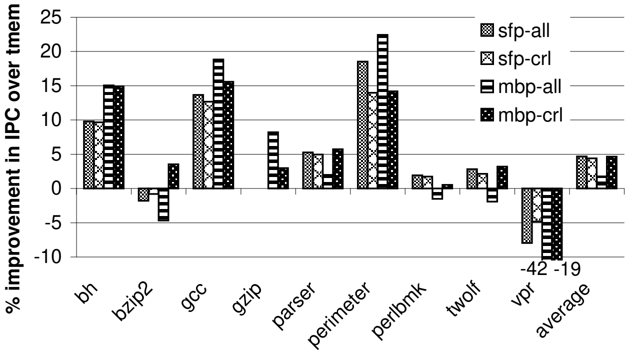
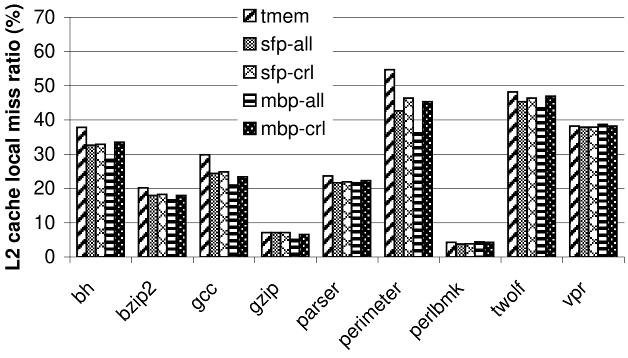
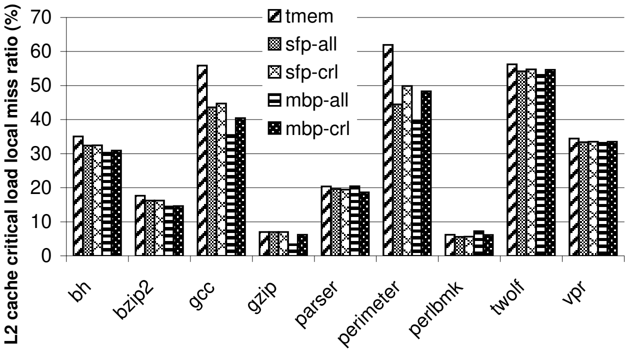


Figure 9. L2 Prefetching: IPC



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 10. L2 Prefetching: Miss Ratio

crl. However, for three other benchmarks, bzip2, perlbmk, and twolf, mbp-all decreases performance, while the criticality based mbp-crl produces performance improvements. Across all four prefetching schemes, mbp-crl is the only scheme that increases performance for bzip2 and it also shows the most performance gains for parser and twolf.

To better understand these results, we look at both cache pollution and resource (L2 MSHR entries and memory bus) constraints. If pollution due to prefetching is a problem, it will lead to an increase in the number of regular load misses compared to tmem. However, our simulations reveal that none of the prefetching schemes increase the overall or critical load L2 local miss ratio considerably (see Figure 10). Using the reference traces collected before, we find this to be true for cache sizes ranging from 128KB to 16MB. Hence, we infer that pollution is not a problem at the L2 cache level.

Figure 11 shows the average completion delay of loads that access main memory. The completion delay of loads is the time interval between when a load's operands are ready and when the load completes. Ideally, the completion delay of loads that access main memory should be 310 cycles. Full MSHRs and busy buses at both the L1 and L2 cache levels are the primary factors behind the increase in completion delay of loads.

From Figure 11, we see that prefetching for vpr increases the resource constraints by increasing the average completion delay of memory accesses compared to tmem. Furthermore, prefetching does not produce a significant decrease in the overall or critical load L2 local miss ratio for vpr. Hence, prefetching is not beneficial for vpr.

For the benchmarks, gcc, and perimeter, for which the locality based scheme, mbp-all, produces considerably higher performance improvements than the other prefetching schemes, we see that prefetching does not introduce any resource constraints. For gcc and perimeter, prefetching decreases the average completion delay of memory accesses and hence does not delay regular loads. Since neither pollution nor resource constraints are a problem for these benchmarks, the most aggressive prefetching scheme—mbp-all—produces the most performance gains. For gzip, both MBP schemes increase the average completion delay of memory accesses, but the number of L2 cache misses for gzip is very small to begin with. The MBP schemes further reduce the number of loads going to main memory and hence produce performance gains in spite of the increase in average completion delay of memory accesses.

For the three benchmarks, bzip2, parser, and twolf, for which the criticality based scheme, mbp-crl, produces the most performance benefits, resource constraints do pose a problem. As can be seen from Figure 11, these are the only benchmarks (other than gzip and vpr) for which there is an increase in the average completion delay of memory accesses due to prefetching. Hence, carefully choosing what to prefetch becomes important for these benchmarks. For parser, and twolf, restricting the number of prefetches using either the locality based sfp-all, or the criticality based mbp-crl produces similar performance. However, for bzip2, mbp-crl has a lower critical load miss rate than sfp-all, and hence produces performance benefits significantly higher than sfp-all. Even with a smaller number of MSHR entries,

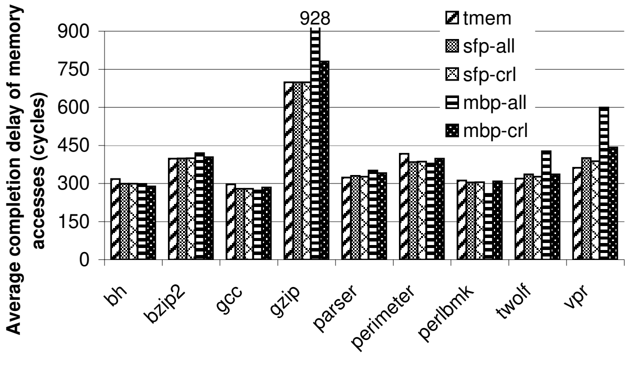


Figure 11. L2 Prefetching: Bandwidth

24 instead of 128, the results are qualitatively similar.

In summary, at the L2 cache level, we find that cache pollution is not a significant problem. If resource constraints are also not a problem, then the most aggressive locality based prefetching scheme, mbp-all, does best. For the benchmarks for which resource constraints are a problem, the selective prefetching scheme based on criticality, mbp-crl, gives the most performance benefits. However, the performance improvements of criticality based prefetching compared to locality based prefetching are small, and may not be worth the added complexity of detecting critical loads.

5 Related Work

Previously, researchers have attempted to improve memory system performance by partitioning memory accesses in several different ways. Some classify memory accesses into those that have spatial locality and those that have temporal locality [15, 7, 5] while others [20] mark load instructions that account for most of the data cache misses as "troublesome". The drawback with these approaches is that they tend to focus on the quantity and not the quality/nature of cache misses. In our work, we evaluate partitioning loads as critical and non-critical based on their latency requirements.

There are several definitions of criticality in previous literature. Calder et al [2] define instructions on the longest path at any point of time as critical, and selectively value predict only critical instructions. Fisk and Bahar [4] use our previous work as motivation to count the number of dependent instructions that attach to a load, and monitor the processor issue rate while a cache miss is being serviced, to classify loads as critical. This scheme looks at only the number and not the type of dependent instructions to classify loads as critical and performs this check only on L1 cache misses.

Various studies examine cache organizations to retain data according to a cache block's temporal locality [15, 7, 8]. Fisk and Bahar use a Non-Critical Buffer (NCB) to hold data classified as non-critical and attempt to free up primary cache space for critical data. They report maximum performance improvements of 4%, but fail to provide insights on why they are unable to achieve further performance gains.

Multi-block prefetching is motivated by investigations of optimal statically-determined cache block fetch sizes [14]. Johnson et al. [9] study dynamically varying the fetch size by fetching bigger cache lines for blocks with good spatial locality. Our mbp-crl scheme prefetches bigger cache blocks whenever a cache miss is generated by a critical load.

6 Conclusion

Current caches are designed to exploit locality of accesses and are unaware of the criticality of loads. In this paper, we present a hardware scheme to estimate the criticality of loads by keeping track of a load's dependence chain as well as the processor's ability to find and execute instructions independent of a load. This scheme classifies 30% of dynamic loads as critical, on average. If all critical loads are guaranteed to hit in the L1 (L2) cache, we see performance benefits of 40% (43%) on average, over a traditional memory system. After establishing the potential for exploiting criticality, we compare the two properties, locality and criticality, in the context of several caching and prefetching schemes to answer the question: In practice, can criticality beat locality?

We find that our criticality based cache organization scheme that uses the critical cache is unable to significantly reduce the critical load miss rate compared to a traditional memory system at both the L1 and L2 cache levels. This is because, the working set of critical loads is large and even retaining critical data alone, fails to reduce competition for cache space.

Our criticality-based prefetching scheme, mbp-crl, selectively prefetches bigger cache lines on critical load cache misses. At the L2 cache level where pollution is not a problem, mbp-crl reduces resource constraints introduced by prefetching, and achieves lower critical load miss ratios compared to a traditional memory system for the benchmarks (bzip2, parser and twolf) that have resource constraint problems. Hence, for these benchmarks, mbp-crl achieves higher performance than locality based prefetching schemes. However, the performance gains are small and may not be worth the added complexity of detecting critical loads. For three other benchmarks, gcc, gzip, and perimeter, resource constraints are not a problem and hence, the most aggressive locality based prefetching scheme, mbp-all, performs best.

Our results indicate that it is very difficult to build memory hierarchies that violate locality to exploit criticality.

Criticality based techniques that co-exist with, and supplement locality, such as pre-executing the backward slices of critical loads [21], may produce higher performance, but require further research.

Acknowledgements

This work supported in part by NSF CAREER Award MIP-97-02547, DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 and CDA-95-12356, Duke University, and Intel Corporation. We thank the anonymous reviewers and Steve Reinhardt for comments and suggestions on this work. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

References

[1] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors-the SimpleScalar Tool Set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[2] B. Calder, G. Reinman, and D. Tullsen. Selective Value Prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 64–75, June 1999.

[3] T. F. Chen and J. L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[4] B. R. Fisk and R. I. Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency. In *Proceedings of the IEEE International Conference on Computer Design*, October 1999.

[5] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of ACM International Conference on Supercomputing*, pages 338–347, July 1995.

[6] D. Grunwald and D. Joseph. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[7] L. K. John and A. Subramanian. Design and Performance Evaluation of a Cache Assist to Implement Selective Caching. In *Proceedings of the IEEE International Conference on Computer Design*, pages 510–518, October 1997.

[8] T. L. Johnson and W. W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[9] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 57–64, December 1997.

[10] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[11] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[12] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, June 1998.

[13] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, Massachusetts, October 1992.

[14] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 160–169, May 1990.

[15] J. A. Rivers and E. S. Davidson. Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 1, pages 154–163, August 1996.

[16] A. Roger, M. Carlisle, J.Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Sytems*, 17(2), March 1995.

[17] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[18] G. Sohi. Instruction Issue Logic for High Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[19] S. T. Srinivasan and A. R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 148–159, December 1998.

[20] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, December 1995.

[21] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, June 2000.