A Fully Associative Software-Managed Cache Design

Erik G. Hallnor and Steven K. Reinhardt
Advanced Computer Architecture Laboratory
Dept. of Electrical Engineering and Computer Science
The University of Michigan
1301 Beal Ave.
Ann Arbor, MI 48109-2122
{ehallnor,stever}@eecs.umich.edu

ABSTRACT

As DRAM access latencies approach a thousand instruction-execution times and on-chip caches grow to multiple megabytes, it is not clear that conventional cache structures continue to be appropriate. Two key features—full associativity and software management—have been used successfully in the virtual-memory domain to cope with disk access latencies. Future systems will need to employ similar techniques to deal with DRAM latencies. This paper presents a practical, fully associative, software-managed secondary cache system that provides performance competitive with or superior to traditional caches without OS or application involvement. We see this structure as the first step toward OS- and application-aware management of large on-chip caches.

This paper has two primary contributions: a practical design for a fully associative memory structure, the *indirect index cache* (IIC), and a novel replacement algorithm, *generational replacement*, that is specifically designed to work with the IIC. We analyze the behavior of an IIC with generational replacement as a drop-in, transparent substitute for a conventional secondary cache. We achieve miss rate reductions from 8% to 85% relative to a 4-way associative LRU organization, matching or beating a (practically infeasible) fully associative true LRU cache. Incorporating these miss rates into a rudimentary timing model indicates that the IIC/generational replacement cache could be competitive with a conventional cache at today's DRAM latencies, and will outperform a conventional cache as these CPU-relative latencies grow.

1. INTRODUCTION

With each succeeding generation of microprocessors, advances in instruction-execution rates further outpace improvements in DRAM access latencies. For the earliest microprocessors, a DRAM access easily fit into a CPU cycle; today, a DRAM access today represents a hundred or more CPU cycles and several hundred instruction-execution opportunities. This processor-memory gap has largely been addressed through the use of caches. Here again, steady, rapid advances in technology have brought about significant changes. Early microprocessor caches were off-chip and a few kilobytes in size, while we now see on-chip SRAM caches in excess of one megabyte [10]. We believe that the conjunction of these two trends—miss latencies approaching a thousand instruction-execution times and multi-megabyte on-chip caches—should spur a re-examination of how secondary (level-two) caches are organized and managed.[1]

In many ways, the cache-DRAM relationship is becoming similar to that between DRAM and disk storage [7][20]. In the latter case, systems employ two mechanisms to aggressively minimize the impact of disk accesses: software-based replacement and prefetching policies and fully associative address mapping. Software-based policies enable relatively sophisticated algorithms for maximizing storage efficiency and prefetching accuracy. Full associativity allows the replacement algorithm to retain the most important pages without regard to address mapping conflicts.

This paper presents a practical, fully associative, software-managed secondary cache system whose performance is competitive with and often superior to more conventional alternatives. We use the term "software managed" to describe a cache in which software explicitly controls the placement of data in the cache, determining precisely which block will be evicted to make room for new data. Cache-management software may execute on the primary CPU itself, perhaps as one thread of a multithreaded CPU, or on a dedicated controller.

Our system consists of two parts: a hardware design, the *indirect index cache* (IIC), and a replacement algorithm, *generational replacement*. The IIC provides a practical implementation of a fully associative data store using indirection, mapping physical addresses to data array indices in a manner similar to page-table-based virtual-to-physical address translation. Unlike virtual address translation, the IIC is transparent to the operating system. Also unlike virtual memory, the IIC initiates miss handling in hardware, keeping software off the critical path of main-memory accesses. Our replacement algorithm, generational replacement, is designed to tolerate the filtered address stream provided by the primary caches while operating with low state-management overhead.

Our preliminary evaluation of the IIC and generational replacement employs a set of Windows NT system address traces from Intel and an S/390 trace from IBM. We find that, for a unified one-megabyte secondary cache, our proposed design provides miss rates 8% to 85% lower than a traditional four-way associative LRU cache, even with profile-based page coloring. Our design also compares favorably to a four-way LRU cache with a 256-entry victim buffer. A simple timing model incorporating estimated IIC

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA 00 Vancouver, British Columbia Canada
Copyright (c) 2000 ACM 1-58113-287-5/00/06-107 $5.00

1. The miss latencies and capacities of primary (level-one) caches have undergone a less dramatic transformation, since the size of these caches is constrained by the processor cycle time and the miss latency is kept low by the now-common presence of a secondary cache.

overheads indicates that the IIC with generational replacement is competitive with a more traditional organization when main-memory access times are 100-200 cycles larger than secondary cache hit times.

In the long run, the primary advantage of fully associative data stores is the flexibility they provide to upper-level management algorithms. Of course, the sophisticated applications, compilers, and operating systems that leverage this flexibility will not spring into existence overnight. In the following section (Section 2), we elaborate the potential advantages of a fully associative, software-managed secondary cache.

Section 3 describes the IIC and generational replacement in detail. Section 4 provides a preliminary evaluation of the IIC using generational replacement. Section 5 discusses related work. We present our conclusions and future directions in Section 6.

2. ARGUMENTS FOR SOFTWARE-MANAGED CACHES

In this section, we discuss the potential applications of a fully associative, software-managed secondary cache. We have identified three areas in which such a cache offers an advantage over low-associativity, hardware-managed organizations: it enables more sophisticated replacement algorithms, it reduces the penalty for locking data in the cache, and it enables arbitrary partitioning of the data store.

Many of these applications require support from programmers, compilers, or operating systems. Unfortunately, this software support will not develop until the requisite hardware structures are available. To bypass this chicken-and-egg situation, this paper focuses on the practicality of such a cache in a transparent environment, i.e., with no application or OS support. The purpose of this section is to argue that the performance advantage discussed in Section 4 is only one of many benefits to be derived from a software-managed cache.

Although full associativity and software management are conceptually orthogonal, they have a symbiotic relationship. Software management of a low associativity cache is overkill; the complexity of a software replacement algorithm is generally not justified when there are only a few replacement choices available. On the other hand, the complexity of managing replacement state for a large, fully associative cache is generally beyond the capability of a hardware implementation, unless a trivial policy such as random is used.

The primary motivation for software-managed caches is the ability to apply sophisticated replacement algorithms such as those developed for virtual-memory paging [9][21] to reduce the performance impact of DRAM accesses.[1] Software-based algorithms can apply more resources and more complex analyses to the problem, and more easily incorporate application hints. Of course, these algorithms will likely require modification to work in a secondary-cache environment; exploring this space is a major component of our future work.

Second, a fully associative cache greatly reduces the penalty for locking (pinning) data into the cache. In any cache, locking a block reduces the associativity of the containing set by one. In a two-way associative cache, only one block can be locked per set, and doing so reduces the available associativity by half. In a fully associative cache, the data that can be locked is limited only by the cache capacity, and each locked block reduces the associativity by a negligible amount. The ability to lock data in the cache can be critical to providing reasonable worst-case execution time guarantees, as required by real-time systems. As the performance impact of memory latency increases, avoiding misses in time-critical code will become ever more important. Our proposed design locks the replacement algorithm's data structures in the cache to avoid the complexity of recursive miss handlers (see Section 3).

Third, a fully associative cache can easily be partitioned into arbitrarily sized pieces for use by different processes or threads, effectively creating multiple smaller, dedicated L2 caches. Cache partitioning would directly benefit operating systems that provide quality-of-service guarantees to processes [5]. This feature may also prove useful for avoiding thrashing in multithreaded processors. The cache could also be partitioned functionally, for example, into a demand miss section and a set of prefetch buffers.

1. By extension, a software-managed cache might usefully support many of the less traditional applications of virtual address translation as well, such as shared virtual memory and garbage collection [2].

3. IMPLEMENTATION

This section describes a practical design of a fully associative software-managed cache. Our design comprises two parts: the hardware structure of the cache, which we call an *indirect index cache* (IIC), and a base replacement algorithm, *generational replacement*.

3.1 The Indirect Index Cache

Typical associative cache designs either access all potential data locations in parallel, or use content-addressable memory cells to directly access the desired data. The former requires *n* parallel data array accesses for an *n*-way associative cache, increasing power consumption and limiting the approach to small *n*. The latter approach does not scale well to very large caches.

The IIC reduces the cost of large fully associative caches by borrowing a basic technique from virtual address translation: indirection. In a traditional cache, each tag is statically associated with a single data entry, and simply indicates which block (if any) is stored in that location. An associative cache built in this fashion must search the tags for all potential data locations. In the IIC design, tag entries are not associated with particular data blocks; instead, each tag entry contains a pointer to the data block, i.e., an index into the cache's data array. Because a tag entry can indicate any data array location, the cache is fully associative. Figure 1 illustrates an example IIC design.

Although indirection eliminates the need to search *n* tag entries for an *n*-way associative cache, we are now faced with the problem of locating the correct tag entry for a given address. The IIC's tag array uses a simple hash table organization, similar to a hashed inverted page table [14]. This organization provides reasonably fast lookups with a storage cost proportional to the number of blocks in the data array. The tag array is split into two parts, a primary hash table and secondary storage for chaining, as shown in Figure 1. On each access, the block tag is hashed to generate a primary table index. Our current implementation simply uses the tag modulo the table size. The primary table is associative, so that a single access searches the first few entries of the hash chain (four in the example of Figure 1). Collisions beyond this depth are chained into the secondary hash storage. This split allows a wide, smaller primary table that handles most accesses while only the narrower secondary table need be sized to handle the worst-case hash chain length. It also provides the opportunity for pipelining deeper hash chain searches, or even allowing accesses that hit in the primary table to bypass those requiring longer searches.

The following two subsections compare the IIC design's storage and access time overheads with those of a more traditional cache organization.

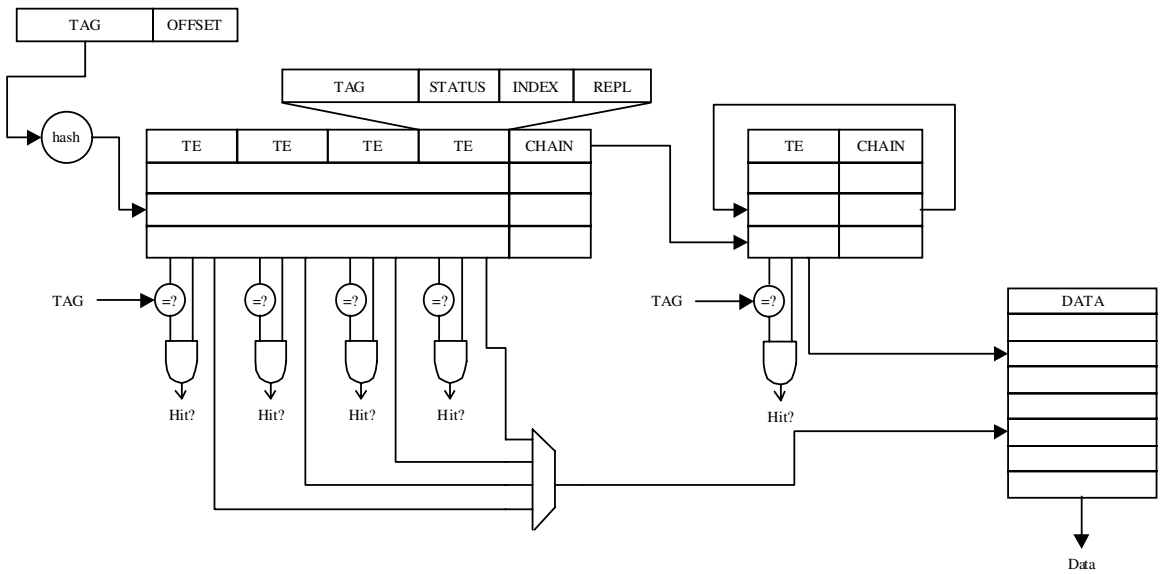

Figure 1. Indirect Index Cache Organization.

3.1.1 Storage overhead

The tag storage overhead of an IIC relative to conventional caches stems from two factors: the data array index and hash chain pointer in each tag entry and need for a larger tag field. The larger tag arises because, unlike a conventional cache, the position of an entry in the IIC's hash-based tag store is not directly related to the corresponding physical address. As a result, the bits used to index the tag array cannot be eliminated from the stored tag. To support software replacement policies, we include a reference bit (set in hardware) and 32 additional bits per tag for policy data storage.

Table 1 shows that for a 1MB cache with 256-byte blocks, assuming a 48-bit physical address, the IIC's tag store has a 198% overhead to that of a 4-way set-associative cache. However, this is an increase of only 32.9 KB, or just over 3.2% of the total storage required by the cache. For 128- and 512-byte blocks the increases are 68.75 KB (207% overhead 6.7% of the cache) and 15.69 KB (188% overhead, 1.5% of the cache), respectively.

	4-way set-associative	**IIC**
Tag size	48–$\log_2$(1M/4) = 30 bits	48–$\log_2$(256) = 40 bits
# of Tags	4096	4096
Status Bits	2 (valid, dirty)	3 (valid, dirty, referenced)
Index Size	N/A	$\log_2$(1M/256) = 12 bits
Chain Ptr	N/A	$\log_2$(1M/256) = 12 bits
Repl. Data size	5 bits per set for LRU (1.25 bits per tag)	32 bits
Tag Entry Size	Tag+Status Bits +Repl Data = 33.25 bits	Tag+Status Bits +Index+Chain Ptr +Repl Data = 99 bits
Tag Store Size	# of Tags*Tag Entry Size = 16.6 KB	# of Tags*Tag Entry Size = 49.5KB

Table 1: IIC Tag Storage Overhead

3.1.2 Access time overhead

The IIC has three primary sources of timing overhead relative to a conventional cache design: additional hit latency due to accessing the tag and data arrays sequentially, additional hit and miss latency due to hash-table-based tag lookups, and additional miss latency due to the overhead of software management.

In most associative cache designs, the tag and data arrays are accessed in parallel. The results of the tag comparison are then used to select one of the *n* data values read from the data array. The IIC would appear to be at a significant disadvantage, since the tag and data accesses must be serialized. However, recent on-chip secondary-cache designs, including the Alpha 21164 [11] and 21364 [10] already serialize their tag and data accesses to reduce power consumption. Checking the tag array first and then accessing only the correct data array bank provides tremendous power savings for a large cache (up to 20W in the case of the 21164 [11]).[1] Given the power constraints of future processors, we expect this approach to be common for on-chip secondary caches; in this case, the IIC's sequential tag/data accesses may not incur a significant access penalty relative to these caches. Note that the IIC design inherently shares the power efficiency of these other sequential-access cache designs.

A second source of overhead is the potential need for multiple tag array accesses to walk the hash chain. By using an 4-way associative primary hash table and moving accessed tag entries to the front, our naïve implementation of the configuration described in section 4 provides average search chain depths of 1.52. Our simple timing model indicates that this overhead is small enough to be offset by potential improvements in miss rates. Optimizing the hash function may further reduce the hash chain lengths.

The third source of overhead is software management. Software-based replacement decisions will require a larger and more variable number of cycles than for a hardware-managed cache. We assume the IIC acts as a transparent, physically addressed cache,

1. This serialization can be avoided by predicting and speculatively accessing the most likely data array element [24, 27]. However, the ability to predict the correct element, and hence the utility of this technique, decreases with increasing associativity.

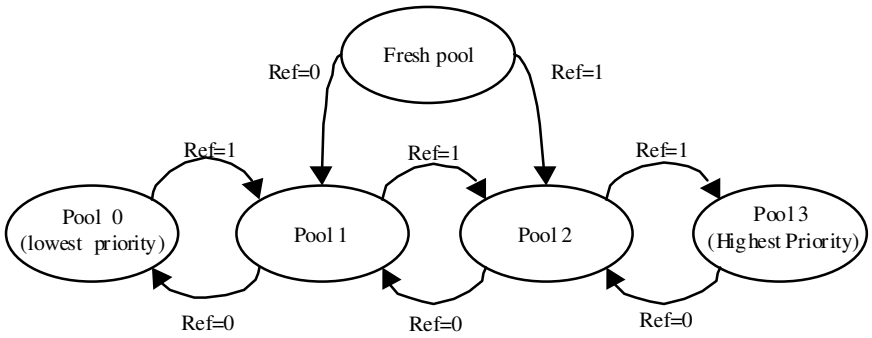

Figure 2. Generational Replacement Algorithm Structure

so (unlike VM) no additional translation is required on a miss.[1] Hardware can thus directly initiate a DRAM access once the tag lookup indicates a miss. The latency of the software handler's replacement decision then overlaps the main memory access latency [12]. Although we are interested in the feasibility of executing cache-management software on the primary CPU, using a dedicated controller tightly coupled to the cache structure will avoid interrupt overheads. We can easily pin the replacement algorithm's code and data structures in the data array, avoiding the need to handle recursive cache misses within the replacement algorithm itself. We assume that replacement processing for multiple misses can be pipelined sufficiently (by multithreading the controller or providing additional hardware support) to avoid causing a bandwidth bottleneck. Finally, the replacement algorithm may maintain a number of unused data blocks as a buffer pool to decouple the timing of the replacement decision from the arrival of data from the memory system.

Of course, the actual latency of replacement decisions depends strongly on the complexity of the algorithm. The following subsection describes one algorithm, generational replacement, which provides competitive performance without significant run-time complexity.

3.2 Generational Replacement

Although the IIC design is independent of the replacement algorithm used to manage it, its utility depends on the existence of a practical algorithm that provides competitive performance without unreasonable overhead. This section presents a novel algorithm we have developed, called *generational replacement*, which is tailored to the needs of a secondary cache. As will be shown in Section 4, generational replacement performs as well as or better than true LRU on all our traces, and better than a standard implementable pseudo-LRU scheme.

In practice, the IIC can use traditional VM paging algorithms such as clock. However, we can improve performance by taking into consideration two significant differences between page replacement and secondary cache replacement. First, primary cache hits are filtered from the secondary cache's observed reference stream. In contrast, because the TLB is usually in front of or in parallel with the primary cache, page reference bits are set according to the unfiltered reference stream. IIC block reference bits are thus poorer than page reference bits as indicators of whether the block is being actively accessed. This effect is compounded by the second difference: secondary cache misses are much more frequent than VM page replacements, so a block that is not referenced between two misses is more likely to be in the program's working set than a page that is not referenced between two page faults. Generational replacement compensates for these effects by incorporating a form of frequency-based hysteresis to overcome the inaccuracies of the IIC reference bits.

3.2.1 Algorithm

As described above, the motivation for generational replacement is to cope with the reduced information provided by secondary-cache reference bits. Specifically, under generational replacement a block that happens not to have its reference bit set during a particular interval is not considered a candidate for replacement if it has been repeatedly referenced in the recent past. Unfortunately, maintaining even approximate reference frequency counters based on reference bits is time consuming. Instead, we group blocks into a small number of prioritized pools. We promote blocks that are referenced regularly into higher-priority pools, and demote unreferenced blocks into lower-priority pools. On a miss, the block to be replaced is chosen from the lowest-priority non-empty pool. We call the algorithm "generational replacement" based on the notion that frequently-accessed blocks are promoted into senior generations, somewhat like long-lived objects are promoted in generational garbage collection [19].

Figure 2 illustrates the algorithm. Each pool is a variable-length FIFO queue of blocks. On a hit, only the block's reference bit is updated. On each miss, the algorithm checks the head of each pool FIFO. If the head block's reference bit is set, it is promoted to the next higher-priority pool; if the reference bit is not set, the block is demoted to the next lower-priority pool. In either case, the reference bit is cleared, and the block is placed at the tail of the new pool's FIFO. To guarantee that a block has an opportunity to be accessed before it is considered for promotion/demotion, each FIFO entry is timestamped when it enters the pool, and a pool is skipped if the head block's residency in the pool is below a fixed threshold. (Time is measured in secondary cache misses to simplify implementation.) New blocks are initially placed in a special "fresh" pool that is not searched by the replacement function to protect these blocks from premature eviction. After meeting the residency time for the fresh pool, they enter the priority chain at the middle.

3.2.2 Implementation

Generational replacement has an inherently small time complexity; the number of operations required on a miss is proportional to the number of priority pools. However, the need for variable-sized FIFOs with per-entry timestamps makes a space-efficient implementation more challenging.

1. The IIC structure could be used for a virtually indexed and tagged cache as well, but given our emphasis on secondary caches, we believe this arrangement is less likely.

The FIFO pools are constructed using doubly linked lists. Two 12-bit block pointers are stored in the 32 bits set aside in each IIC tag entry for replacement data. A pair of pointers for each pool identify the head and tail of the pool's FIFO.

The second potential source of storage overhead is the pool residency timestamp in each record. We minimize the size of these timestamps by keeping in each record a delta from the timestamp of the preceding record. This delta is capped at 255, allowing an 8-bit delta value field. Each pool maintains two full timestamps, corresponding to the head and tail entries. We use the former to determine when the head is eligible for promotion/demotion, and the latter to generate the delta value when a new block is appended to the FIFO.

Overall, the per-block records, provided in the IIC tag array, dominate the storage required. Each pool only requires additional storage for the 2 pointers into the IIC and 2 32-bit timestamps.

4. EVALUATION

We conducted a preliminary evaluation of our design using two sets of instruction traces. The first set was generously provided by Chris Wilkerson of Intel Microcomputer Research Labs. These traces were generated on an Intel Architecture platform running Windows NT 4.0, and include OS and DLL references [18]. We selected five traces that stressed a 1-MB cache: pcdb (a PC database application), draw (a PC drawing program), specweb (a web server trace from SPECweb96), and tpcc and tpcc_long (2 transaction processing server traces). The final trace, oltp1w, was provided by IBM. This trace records level-one cache misses from an S/390 mainframe running an online transaction processing (OLTP) workload. Because level-one hits are already filtered out of the trace, oltp1w stresses the cache significantly more than the Intel traces. Table 2 lists some trace statistics.

Trace	Instr Refs	Data Refs	Data Size (bytes)
draw	47,104,263	29,086,308	2,628,032
pcdb	36,470,324	22,052,355	2,934,400
specweb	88,208,673	44,122,607	10,751,616
tpcc	184,125,212	84,057,750	8,715,776
tpcc_long	40,924,323	21,328,973	3,877,376
oltp1w	15,797,770	16,091,431	51,579,136

Table 2: Trace Statistics

All simulations (other than oltp1w) assume 64KB, 2-way set-associative split primary instruction and data caches with 32-byte blocks. Since oltp1w is a trace of level-one cache misses, no level-one cache simulation was necessary. We did not enforce inclusion between the primary and secondary caches. We fixed the secondary cache size at 1MB (since the traces would not realistically support larger sizes), and studied block sizes from 128 to 512 bytes. Although most current secondary cache designs use block sizes of 64 or 128 bytes, we believe that increasing cache sizes and memory bandwidths will favor larger blocks in the future, so we bias our study in this direction. For the IIC simulations, the 1MB data array size was reduced by the number of blocks needed by the replacement policy.

To characterize the effects of increased associativity on these traces, we simulate caches with a range of associativities from four to full. Figure 3 shows the number of misses for these configurations with two replacement policies: LRU and OPT (Belady's optimal off-line algorithm). As can be seen in the figure, a few cases—oltp1w, draw with 512-byte blocks, and specweb with 256-byte blocks—show a sustained benefit from increasing associativity. Oltp1w's accesses are concentrated in a few sets, resulting in many conflict misses. Although there is some reuse, very high associativity is required to take advantage of it due to the large number of unique blocks that map to the heavily accessed sets. Typically, though, for a given block size and replacement algorithm, the benefits of higher associativity level off beyond 16 ways. On the other hand, for a given block size and associativity, OPT outperforms LRU significantly in almost every case. In fact, for the two tpcc traces, LRU performance degrades slightly as associativity increases. Thus, even when we do not expect a significant benefit from full associativity alone, there is room for improvement via better replacement policies.

To compare the IIC with more traditional approaches to reducing conflict misses, we also simulated the effects of improved page coloring [17] and a 256-entry fully associative victim cache [16]. The NT traces reflect the page-coloring algorithm implemented by the NT kernel. To reflect the potential of improved page coloring, we reassigned physical page numbers in each trace using a simple off-line profile-based algorithm described by Sherwood et al. [25]. These results are optimistic, as the coloring for each trace used the same full trace as its profiling input. (We also implemented on-line bin-hopping [17], but the results were not significantly different.)

Figure 4 compares a subset of the results from Figure 3 with miss counts from alternative cache configurations, including the IIC. The first five bars represent traditional cache designs. The first bar indicates the miss count for our base case, a four-way associative LRU cache, taken from Figure 3. The following two bars show our results for adding page coloring and a victim cache to the four-way configuration, respectively. The fourth and fifth bars are 8-way and 16-way LRU caches (again from Figure 3), pushing the practical limit of conventional associative caches. The following two bars represent practical IIC-based fully associative caches, using two replacement algorithms: two-handed clock (a common pseudo-LRU algorithm used in virtual-memory paging) and generational replacement. The final two bars indicate the miss counts of fully associative caches using true LRU and OPT replacement, respectively; these are idealized caches as the former algorithm is impractical and the latter infeasible.

As can be seen from the figure, the IIC with generational replacement has the best overall performance of any of the implementable cache configurations, with 7% to 85% fewer misses than the traditional 4-way set-associative LRU cache. Its advantage is smallest for the two tpcc traces; in fact, for the 128-byte block size, generational replacement increases the number of misses by 2-3% over the 16-way LRU cache on tpcc and over the 8- and 16-way LRU caches on tpcc_long. On the other hand, generational replacement reduces the miss count relative to 16-way LRU by up to 78% on draw and up to 74% on oltp1w.

While the improved page coloring did reduce the number of misses in the 4-way LRU cache, it still fell short of the performance of the IIC with generational replacement on the non-colored trace. Interestingly, page coloring is useful in conjunction with the IIC to reduce the number of conflicts in the IIC's hash table, shortening the hash chains and thus improving average access time. In our experimental configuration, improved page coloring reduces the number of misses for generational replacement as well, because it reduces the number of misses in our physically indexed L1 cache; with a more typical virtually indexed L1 cache, the number of misses in the IIC would be unaffected. The large victim cache outperformed the improved page coloring in most cases, and (like the 8- and 16-way LRU caches) was competitive with generational replacement on the tpcc traces. However, the victim cache

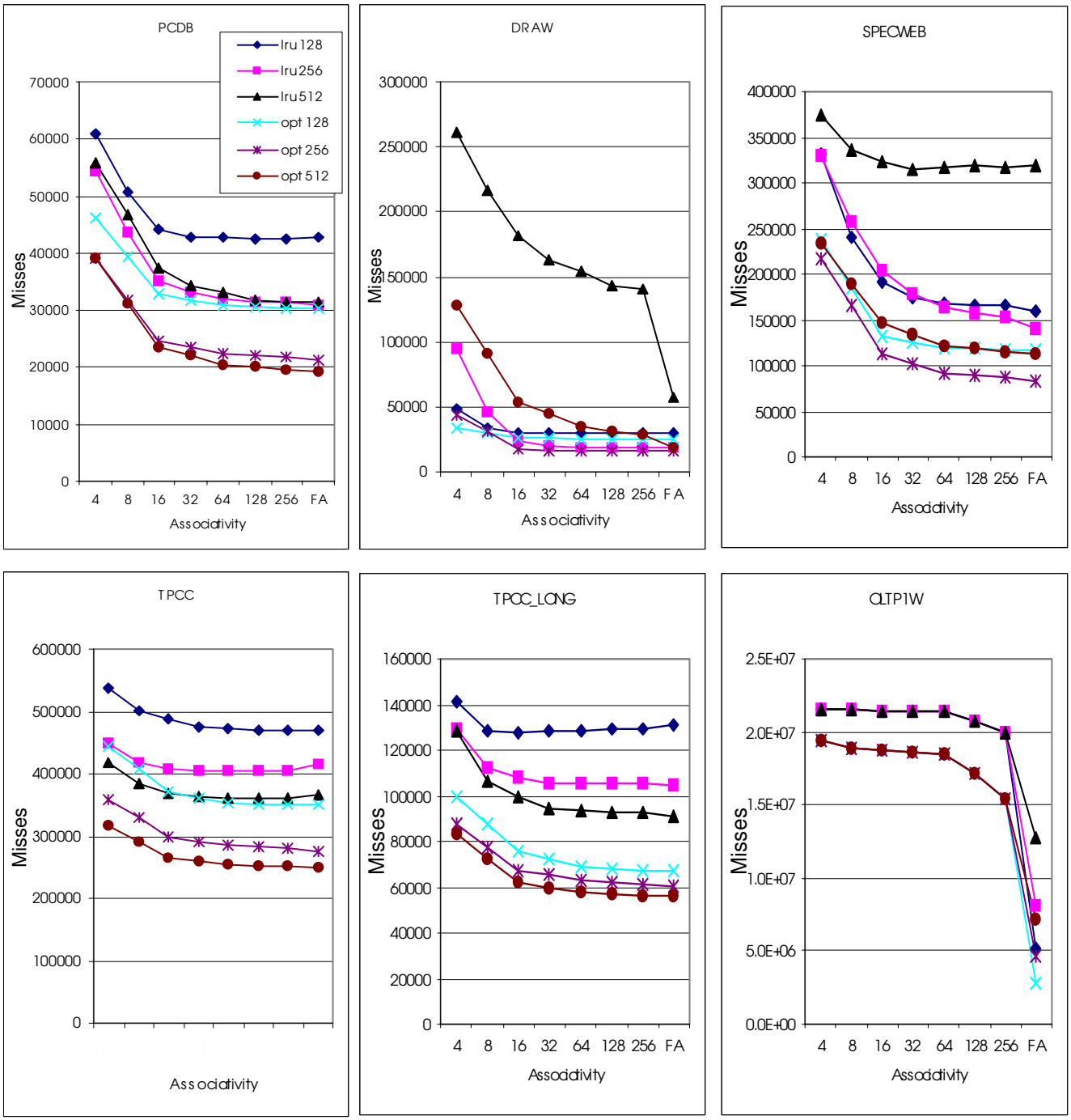

Figure 3. Misses vs. associativity, LRU and OPT

fell short of generational replacement on the other traces, particularly specweb and oltp1w.

Turning to the fully associative caches, generational replacement always performed within 7% of the (impractical) true LRU replacement, and performed significantly better for some block sizes on draw, specweb, and tpcc_long. Results from using random replacement (not shown) indicate that this policy is significantly worse than either generational replacement or LRU, and for the tpcc traces performs worse than 4-way LRU. This result indicates that high associativity alone is not useful without a reasonably effective replacement policy. The pseudo-LRU algorithm, clock, universally performs worse than generational replacement as well, and did particularly poorly on the specweb and tpcc traces.

The absolute miss counts go down as the block size increases from 128 to 256 bytes for all traces except oltp1w on the IIC and for all except oltp1w and draw on the 4-way LRU cache. However, going from 256- to 512-byte blocks causes noticeable jumps in the miss counts for draw, specweb, and oltp1w due to increased con-

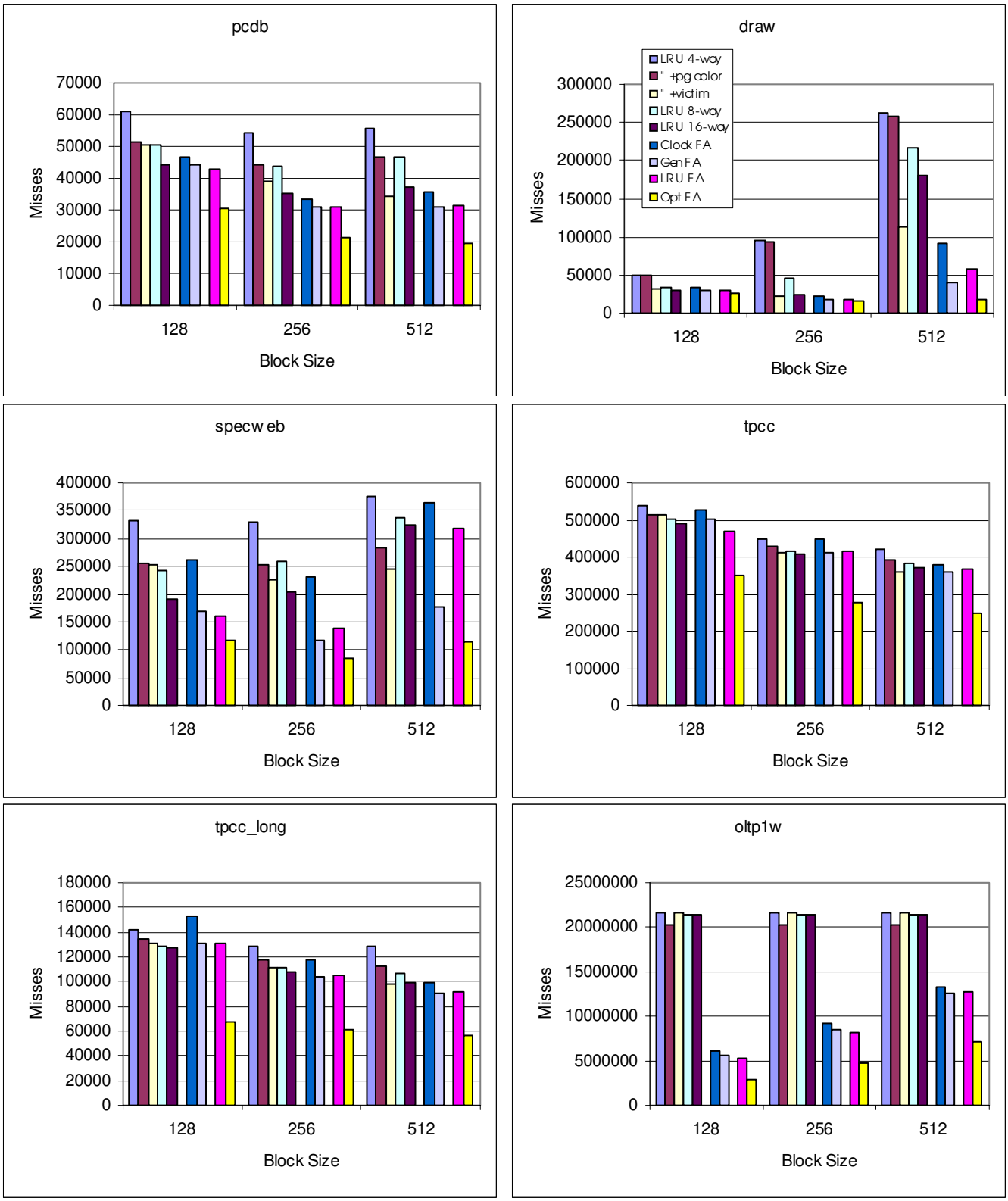

Figure 4. Number of misses for conventional caches, IIC, and fully associative LRU and OPT caches.

Trace	L2 accesses	Block size	LRU 4-way misses	IIC/gen. repl. misses	Improvement in miss count	Hash chain length	Break-even miss latency (cycles)
pcdb	929,577	128	60,858	44,423	27.0%	1.39	84.9
		256	54,378	31,137	42.7%	1.45	66.0
		512	55,825	30,886	44.7%	1.5	64.0
draw	1,331,662	128	49,054	30,569	37.7%	1.71	130.4
		256	94,627	18,453	80.5%	1.53	36.4
		512	261,315	39,378	84.9%	1.45	18.4
specweb	3,971,648	128	332,592	169,134	49.1%	1.53	45.6
		256	329,294	117,505	64.3%	1.44	36.2
		512	374,547	176,404	52.9%	1.34	35.7
tpcc	3,978,301	128	536,806	500,103	6.8%	1.71	172.0
		256	448,300	411,298	8.3%	1.59	163.3
		512	418,985	357,678	14.6%	1.46	96.2
tpcc_long	1,732,243	128	141,667	131,617	7.1%	1.68	277.6
		256	129,049	104,032	19.4%	1.58	112.8
		512	128,804	90,367	29.8%	1.48	73.2

Table 3: Break-even Analysis

tention for the smaller number of unique tags. The combination of full associativity and generational replacement seems to cope with this pollution much more readily than any of the feasible alternatives. (Note the change in performance of the fully associative LRU cache for specweb.) Larger block sizes also reduce the relative tag overhead of the IIC. As cache capacities increase to multiple megabytes, larger block sizes will become more attractive, increasing the IIC's benefit.

Still, these results do not indicate a performance advantage for the IIC since they do not incorporate timing information and do not account for the IIC's timing overheads. To provide a very rough feel for the performance tradeoff, we applied a simple timing model to our results. We assume that the 4-way LRU cache has a 10-cycle hit time and a miss penalty of an additional *m* cycles. We gave the IIC a 1-cycle penalty per hash chain access, counting the primary table as access, on both hits and misses. Thus in the IIC hits cost (10 + <hash chain length>) cycles and misses take (10 + <hash chain length> + *m*) cycles. We then calculated the break-even value of *m* where the reduced miss rate of the IIC would compensate for the increased access time. This value is listed for each benchmark and block size in the final column of Table 3.[1] For nearly all the benchmarks and block sizes, these values are within—and in some cases well under—the number of cycles seen by current processors on a DRAM access. Note that if we increase the base hit time of the LRU cache, or if optimizations (such as better hashing) to reduce IIC tag access time are effective, these break-even points will decrease further.

5. RELATED WORK

The growing resemblance of DRAM accesses to virtual-memory page faults has been noted before by Machanick [20] and Burger [7]. Machanick proposes adopting the virtual-memory model in its entirety, paging from on-chip SRAM to off-chip DRAM. However, even given current trends in technology, it will be a while before the cost of a full OS page fault is as negligible a fraction of a DRAM access as it is with a disk access today. As the IIC demonstrates, we can achieve the benefits of full associativity and software management without adopting some of the larger overheads of the virtual-memory model.

Cheriton et al [8] proposed and built the earliest software-managed caches as part of the VMP project. They focused on using software implementations of cache coherence protocols rather than replacement algorithms. Jacob and Mudge [15] proposed a software-managed secondary cache in which the software handlers perform virtual address translation on cache misses. This scheme places the handlers on the critical path of memory accesses, since the physical address must be determined before the DRAM access can begin. Our IIC design assumes conventional address translation support so that DRAM accesses can be initiated without software intervention. Jacob and Mudge also assumed a direct-mapped cache, so their software handlers have no role in replacement.

A software-managed cache informs software of all misses, as do Horowitz et al.'s informing memory operations [13]. A software-managed cache has the additional advantage that the handler can directly influence the replacement decision made for that miss. Informing memory operations inform the user process directly, while software-managed cache handlers more likely execute in a privileged context. However, most of the applications for informing memory operations presented in [13] could be accomplished equally well without user-level notification.

The DASC cache [23] and the group-associative cache [22], like the IIC, combine an associative tag store with a direct-mapped data array. However, the goal of both these designs is to combine the access time of a direct-mapped cache with the miss rate of a more highly (but not fully) associative cache. Both speculatively access the data array using the address as an index, and potentially reaccess the array if the first access fails but the requested data is present in a secondary location. In the DASC cache, a block can reside in only a limited number of locations. The group-associative cache maintains a small, highly associative tag directory for blocks not in their primary location; these tags contain an index into the data array, so that a block may reside in any location, just as in the IIC. However, the number of tag directory entries is a fraction of the number of cache lines, so only a limited number of blocks can be located in other than their primary location. Both of these cache designs, along with other proposals such as the column-associative cache [1] and victim cache [16], are intended to reduce conflict

1. Due to a bug in our simulator, we were unable to measure exact hash-chain lengths for oltp1w.

misses in a direct-mapped array, but not to provide a flexible data store appropriate for software management.

Operating systems can control data placement in physically indexed caches by choosing physical addresses carefully [17]. These placements can be driven by heuristic algorithms [17], compiler or profile analysis [6][25], or hardware conflict detection [4][25]. While these approaches are effective at spreading accesses more evenly across the cache sets, they do not actually increase the associativity of the cache. Our results show that, for several of the traces we examined, full associativity is far more effective at reducing the number of misses.

Page placement can also be used to pin data in the cache and partition the cache among processes, two of the features described in Section 2. However, a fully associative cache allows cache placement decisions independent from physical memory allocation. For example, page coloring can be used to lock a page at the expense of not caching any other physical memory pages with a conflicting color. This shortcoming could be avoided by adding a cache index field to the TLB [25].

6. CONCLUSIONS

This paper outlines a design for a memory structure—the indirect index cache (IIC) —that provides performance competitive with conventional caches while providing the flexibility of software management. Selectively borrowing techniques from virtual memory, namely indirection and hashed page tables, allows us to create a fully associative store with reasonable access time and size overheads. Avoiding other aspects of virtual memory, particularly the need to invoke a software handler on the critical path of a miss, keeps the IIC's performance competitive with traditional organizations.

We also present a replacement algorithm, generational replacement, that is specifically designed to work with a software-controlled secondary cache. Generational replacement incorporates frequency-based hysteresis to outperform a standard pseudo-LRU algorithm (clock). At the same time, its state-management overheads are low so that handler execution can be overlapped with DRAM access times, and its storage requirements are small enough that all its data can be stored in the tag store of the IIC.

We used system traces to analyze the behavior of an IIC with generational replacement as a drop-in, transparent substitute for a conventional (4-way associative LRU) 1 MB secondary cache. Generational replacement performs very well, providing miss rates within a few percent of true fully associative LRU under all tested configurations, and beating true LRU by 10–40% in a few specific circumstances. This performance translates into a substantial reduction in miss rates (7–85%) relative to a conventional 4-way associative LRU cache. Incorporating these miss rates into a rudimentary timing model, taking into account the access-time overheads of the IIC, indicates that the IIC/generational replacement cache could be competitive with a conventional cache even at today's relative DRAM latencies. The IIC's advantage will grow with succeeding processor generations as the benefit of avoiding a DRAM access increases. The combination of competitive or superior performance as a conventional-cache replacement with potential as an enabling factor for more sophisticated software management indicates that the IIC or a similar structure should be seriously considered for on-chip secondary caches in the near future.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under Grant No. CCR-9734026, by gifts from Intel and IBM, and by a grant from Compaq. Many thanks to Wei-fen Lin for her contributions. Thanks also to Mark Hill for his comments on a draft of this paper.

REFERENCES

[1] A. Agarwal and S. D. Pudar. Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 179-190.

[2] A.W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 96-107.

[3] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, Vol. 5, No.2, 1966.

[4] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 158-170.

[5] J. Bruno, E. Gabber, B, Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. *USENIX 1998 Annual Technical Conference*, June 1998, pp. 235-246.

[6] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum and M. S. Lam. Compiler-Directed Page Coloring for Multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp. 244-255.

[7] D. Burger. *Measuring and Reducing the Performance Impact of Memory Traffic*. PhD Thesis, University of Wisconsin-Madison, November 1998.

[8] D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen. The VMP Multiprocessor: Initial experience, Refinements and Performance Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988, pp. 410-421.

[9] G. Glass and P. Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *Proceedings of SIGMETRICS 1997*, May 1997, pp. 115-126.

[10] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14):12-15, Oct. 26, 1998.

[11] L. Gwennap. Digital leads the pack with 21164. *Microprocessor Report*, 8(12):1-6, Sept. 12, 1994.

[12] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 274-285.

[13] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 252-263.

[14] J. Huck and J. Hays. Architectural Support for Translation Table Management in Large Address Space Machines, In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 39-50.

[15] B. Jacob and T. Mudge. Software-Managed Address Translation. In *Proceedings of the 3rd Annual International Symposium on High-Performance Computer Architecture (HPCA)*, February 1997.

[16] 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 Symposium on Computer Architecture*, May 1990, pp. 364-373.

[17] R. Kessler and M. Hill. Page Placement Algorithms for Large Real-indexed Caches. *ACM Transactions on Computer Systems*, 10(4), Nov. 1992.

[18] 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*, July 1998.

[19] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. CACM 26:6, June 1983, pp. 419-429.

[20] P. Machanick, P. Salverda, and L. Pompe. Hardware-Software Trade-Offs in a Direct Rambus Implementation of the RAMpage Memory Hierarchy. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998, pp. 105-114.

[21] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995, p. 79-95.

[22] J.-K. Peir, Y, Lee, and W. W. Hsu. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998, pp. 240-250.

[23] A. Seznec. DASC Cache. In *Proceedings of the 1st Annual International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 1995, pp. 134-143.

[24] K. So and R. Rechtschaffen. Cache Operations by MRU Change. *IEEE Transactions on Computers*, 37:6, June 1988, pp. 700-708.

[25] T. Sherwood, B. Calder, and J. Emer. Reducing Cache Misses Using Hardware and Software Page Placement. *Proceedings of the International Conference on Supercomputing*, June 1999.

[26] A. S. Tanenbaum. Modern Operating Systems. Prentice Hall, Englewood Cliffs, NJ, p. 111 (1992).

[27] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16:2, April 1996, pp. 28-41.