

Optimal Replacement Is NP-Hard for Nonstandard Caches

Mark Brehob, Stephen Wagner, Eric Torng, and Richard Enbody, *Member, IEEE*

Abstract—When examining a new cache structure or replacement policy, the optimal policy is a useful baseline. In this paper, we prove that finding the optimal schedule is NP-hard for any but the simplest of caches, and that no polynomial-time approximation scheme exists for this problem unless $P = NP$.

Index Terms—Cache, optimal cache replacement policy, interval scheduling, approximation algorithm, victim cache, skew cache, multilateral cache.

1 INTRODUCTION

ONE of the greatest challenges of modern computer architecture is coping with the disparity between processor and memory speeds. Standard caches (direct-mapped, set-associative, and fully-associative caches) have improved memory hierarchy performance to partially compensate for this disparity, but, as this disparity grows, new ideas are needed. Many newer caches, including victim caches [9], assist caches [5], skew caches [13], and “smart” caches [7], [8], [16], [18], [19], have been proposed as possible improvements to standard caches and some have been implemented.

Unfortunately, we do not fully understand the behavior of these new caches because many tools that have been used to analyze standard caches have not been adapted to these new caches. This complicates the evaluation and comparison of these caches, as well as any efforts to improve their performance. In this paper, we focus on one specific tool, a polynomial-time optimal replacement strategy.

In 1966, Belady published his seminal paper describing an optimal replacement policy for standard caches [2]. When faced with a replacement decision, Belady’s algorithm evicts the block which will be referenced the furthest in the future. Belady’s algorithm achieves the lowest possible miss rate. Other, faster, algorithms have been developed which also achieve the optimal hit rate [11], [14]. As these algorithms perform the same task and generate equivalent results, we refer to this class of optimal replacement algorithms as OPT.

OPT is an offline algorithm (that is, it requires future knowledge), so OPT cannot be implemented in an actual cache. However, it is an important analytic tool for use in comparative architectural studies. For example, it has been used as a benchmark to judge replacement algorithms [15], [17], [18], [19], to show how much performance headroom might exist for a new replacement algorithm [15], to compare caching schemes independent of replacement algorithms [16], and to judge minimum bandwidth requirements [4]. It has also been used as part of a scheme for identifying memory references with significant locality [19].

Because none of the existing OPT algorithms extend to new caches such as victim caches and skew caches, cache architecture

- M. Brehob is with the Department of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122. E-mail: brehob@umich.edu.
- S. Wagner, E. Torng, and R. Enbody are with the Department of Computer Science and Engineering, 3115 Engineering Bldg., Michigan State University, East Lansing, MI 48824-1226. E-mail: {torng, wagners5, enbody}@cse.msu.edu.

Manuscript received 30 June 2000; revised 28 Mar. 2003; accepted 22 Apr. 2003.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112377.

researchers have been searching for new polynomial-time optimal replacement strategies for these new caches. Until now, they have been stymied in their efforts and have settled for using tractable but suboptimal replacement strategies instead of an optimal strategy, e.g., [16]. We now show that all computer architecture researchers may be forced to make this compromise because no polynomial-time optimal replacement strategy exists for most nonstandard caches unless $P = NP$.

1.1 Companion Caches

Every cache we are aware of has two major components: its *structure* and its *replacement policy*. The structure of a cache describes the physical properties of the cache (such as cache line size, number of cache lines, etc.) and the legal locations within the cache for any block. The replacement policy describes under what circumstances a block is placed into the cache, which block or blocks are removed, and into which legal location the block is placed. For example, the set-associative cache is a cache structure and LRU is a replacement policy.

We now introduce the term *Companion Cache Structure (CCS)*. A CCS consists of two components: a set-associative or direct-mapped cache and a fully-associative cache. We refer to the direct-mapped/set-associative component as the “main cache” and the fully-associative component as the “companion buffer.” Both components must be able to store any arbitrary cache block. A *degenerate CCS* is one in which the main cache is fully-associative or the companion buffer has size zero. In the first case, the CCS reduces to a single fully associative cache. In the second case, the CCS reduces to just the main cache, which is set-associative or direct-mapped.

Many common caching schemes have a CCS underlying them. Cache structures such as a victim cache or an assist cache, together with its associated main cache, are exactly CCSs. Most of the “smart” caches are also CCSs. In addition, we demonstrate that skew caches and certain multilevel caches contain CCSs.

In this paper, we show that the problem of optimally managing a nondegenerate CCS is NP-hard. We assume that there are no placement restrictions other than those that are part of the CCS. For example, the victim caching scheme, as originally defined by Jouppi [9], consisted of both a structure (a direct-mapped cache and the victim cache itself) as well as a replacement policy. In this case, our results apply to the structure of a direct-mapped cache and the victim cache, ignoring any limitations imposed by Jouppi’s replacement policy.

1.2 Related Terms

While CCSs encompass a large number of useful cache organizations, it is important to put CCSs in the context of other terms and concepts. The *multilateral cache* [16] is perhaps most closely related to the CCS. A multilateral cache consists of multiple cache structures which are conceptually on the same level of the memory hierarchy and which can store the same data. In general, multilateral caches contain an underlying CCS, although it is not always obvious.

A *multilevel cache* can also contain an underlying CCS. Say the L1 cache is fully-associative and the L2 cache is direct-mapped, and the L2 cache is *not required* to contain a superset of the items of the L1 cache. While these two caches may have very different access times, they can be modeled as a single CCS. Thus, our results apply to multilevel caches.

1.3 Problem Definition

To facilitate our formal definition of the cache management problem, we define the following two terms: *bypassing* and *internal reorganization*. Standard caching schemes require that each block be placed in the cache when it is referenced. If this requirement is not in force, the cache is said to allow *bypassing*. Another variation is to

allow a block to move from one legal location to another without evicting it. We call this *reorganization*. Many modern caching schemes allow for reorganization to some extent; in fact, it is at the heart of the replacement schemes for both victim and assist caches. Notice that allowing bypassing or reorganization cannot increase the miss rate of an optimal algorithm. Further note that reorganization does not affect the miss rate in a standard cache as all legal locations are equivalent.

We now formally define the offline cache management decision problem. An input instance consists of a cache structure, an ordered list of memory references s , a mapping g describing where each reference can be placed in the cache, and an integer bound K on the number of desired hits. Other details such as whether or not reorganization or bypassing is allowed are defined independent of any specific instance. The question to be answered is whether or not the sequence of requests can be legally serviced with at least K hits.

Legal processing is defined as follows: The cache is initially empty. The memory references in s are then processed in order. If the current memory reference is in the cache, this is referred to as a hit. If reorganization is allowed, the system may reorganize items within the cache subject to the limitations of the mapping g , but may not place any new items in the cache. If the current memory reference is not in the cache, this is referred to as a miss. If reorganization is allowed, the system may reorganize items within the cache subject to g . If bypassing is allowed, the system may choose to not place the current memory reference into the cache. If bypassing is not allowed, the current reference must be placed in the cache, resulting in the eviction of the item it dislodges.

1.4 Paper Outline

The bulk of this paper is a formal proof that the offline cache management decision problem is NP-hard when the cache is a *CCS* where the companion buffer has one location, the main cache is direct-mapped, bypassing is allowed, and reorganization is not. We then describe several generalizations of this result (proofs are omitted due to space limitations).

- The problem is still NP-hard if the companion buffer is of a size greater than one and/or the main cache is set-associative.
- No polynomial-time approximation scheme exists for a *CCS* cache unless $P = NP$.
- The problem is NP-hard whether or not bypassing is allowed.
- The problem is NP-hard whether or not reorganization is allowed.
- Skew and certain multilevel caches are NP-hard to schedule optimally.

The net result of all of these proofs is that using optimal scheduling to evaluate replacement algorithms or cache structural changes is not viable for any but the simplest cache structures. This result is important so that cache researchers and designers do not look fruitlessly for a polynomial-time optimal replacement algorithm. If their cache structure has an underlying *CCS*, they will have to live without that analytic tool.

2 PROOFS

This section is divided into three parts. In Section 2.1, we describe an interval scheduling problem that we will use in our proof. In Section 2.2, we provide a formal proof of NP-hardness for a very restricted problem—when the main cache is direct-mapped, the companion buffer is of size one, no internal reordering is allowed, and bypassing is allowed. We call this restricted case the Fundamental Companion Cache Scheduling (FCCS) problem. In Section 2.3, we describe several generalizations of the FCCS result.

We formally define the FCCS problem as follows:

Definition 2.1 Fundamental Companion Cache Scheduling (FCCS). *INSTANCE:* A direct-mapped cache D , a companion buffer consisting of a single location C_1 , a set of memory items (cache blocks) M , a mapping $g : M \rightarrow D$ indicating in which location in the direct-mapped cache each memory item can be stored, a sequence s of requests for memory items from M , and a positive integer $K \leq |s|$.

QUESTION: Can the sequence s of memory items be legally serviced with at least K hits where no reorganization is allowed, but bypassing is allowed?

To prove that FCCS is NP-hard, we will be working with the known NP-complete problem 3-Occ-Max-2SAT [3]. 3-Occ-Max-2SAT is a restricted form of Max-2SAT where each variable occurs at most three times. More formally, we define the problem as follows:

Definition 2.2 3-Occ-Max-2SAT. *INSTANCE:* A set of variables U , a collection C of clauses over U such that each clause $c \in C$ has $|c| = 2$, and a positive integer $K \leq |C|$. Furthermore, the total number of occurrences of any variable $u \in U$ in C is at most three.

QUESTION: Is there a truth assignment for U that simultaneously satisfies at least K of the clauses in C ?

2.1 Interval Scheduling and Caching

We initially assume that the cache allows bypassing; thus, an item is put in the cache only if it will remain in the cache until its next access. With bypassing, there is a strong correlation between cache management problems and interval scheduling problems.

In interval scheduling, there is a set of machines and a set of intervals (jobs), each of which has a fixed start time and a fixed end time. A machine can work on only one interval at a time. An interval must be processed continuously from its start time to its end time on a single machine to be completed. Each interval can be processed only by a given subset of the machines. The goal is to maximize the number of intervals completed.

If every interval can be placed on any machine, interval scheduling is solvable in polynomial time [6]. We focus on *restricted interval scheduling* problems where each interval can be placed only on a subset of machines. While versions of restricted interval scheduling can be solved in polynomial time, including those that correspond to set-associative cache management problems, the general case of restricted interval scheduling is NP-complete [1], [10].

To show that the FCCS problem is NP-complete, we first show that the Fundamental Companion Interval Scheduling problem (FCIS), a special case of restricted interval scheduling, is NP-complete. We then reduce FCIS to FCCS. FCIS is formally defined as follows:

Definition 2.3 Fundamental Companion Interval Scheduling (FCIS). *INSTANCE:* A set G of m machines G_1, \dots, G_m , a “companion” machine C_1 , an integer Q , and a set I of n intervals (s_i, f_i, σ_i) , where $s_i \in \mathbb{Z}^+$ is the start time of interval i , $f_i \in \mathbb{Z}^+$ is the end time of interval i and $\sigma_i \in G$ is the machine on which interval i can be scheduled. Any interval can be scheduled on the companion machine.

QUESTION: Can at least Q of the intervals be legally scheduled?

2.2 Formal NP-Hard Proof for FCCS

Theorem 2.1. *FCIS is NP-complete.*

Proof. We reduce 3-Occ-Max-2SAT to FCIS. Let $U = \{u_1, u_2, \dots, u_a\}$, $C = \{c_1, c_2, \dots, c_b\}$, and K be an instance of 3-Occ-Max-2SAT. The value of Q for FCIS is $3a + b + K$. For each variable $u_i \in U$, create two machines, M_{u_i} and \overline{M}_{u_i} , both in G , and create four intervals $(i-1, i, M_{u_i})$, $(i-1, i, \overline{M}_{u_i})$, $(0, a+b, M_{u_i})$, and $(0, a+b, \overline{M}_{u_i})$. We call the first two intervals “short intervals” and the second two intervals “long intervals.” The a variables generate a total of $4a$ intervals. For each clause $c_j \in C$, create the

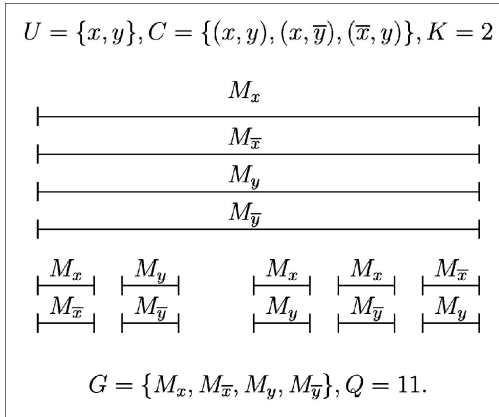


Fig. 1. Example reduction from 3-Occ-Max-2SAT to FCIS.

following two intervals: If variable u_i appears in clause c_j in its positive form, we create the interval $(a + j - 1, a + j, M_{u_i})$. If u_i appears in the negative form, we create the interval $(a + j - 1, a + j, M_{\bar{u}_i})$. The b clauses generate a total of $2b$ “clause” intervals. Altogether, a total of $4a + 2b$ intervals and $2a$ machines are created and both sets can clearly be computed in time polynomial in the original input size.

An example transformation is shown in Fig. 1. The intervals form an instance of FCIS. The intervals can be partitioned into three groups: the long intervals, the short intervals, and the clause intervals. The key observations about this reduction are that the long and short intervals are used to enforce a truth assignment to the variables and pairs of schedulable clause intervals correspond to the satisfiable clauses.

Suppose we can satisfy K of the clauses in the 3-Occ-Max-2SAT instance. We now show that we can schedule $3a + b + K$ of the intervals. Consider each variable $u \in U$. If u is true, we use machine $M_{\bar{u}}$ to schedule a long interval corresponding to u and use machines M_u and C_1 to schedule the two short intervals corresponding to u . If u is false, the roles of machines $M_{\bar{u}}$ and M_u are reversed. Exactly $3a$ of the long and short intervals will be scheduled.

Now, consider each pair of clause intervals. If the corresponding clause c is satisfied in the 3-Occ-Max-2SAT problem, one of the literals in c must be true. Without loss of generality, we can assume that this literal is l . Because l is true, the long interval that could be placed on M_l was not scheduled and machine M_l is free. Therefore, we can schedule the pair of intervals that correspond to clause c using machines M_l and C_1 . If the clause is not satisfied, only one interval can be scheduled (on C_1). It is important to note that, for each variable u , the clause intervals either use machine M_u or machine $M_{\bar{u}}$, but not both. A total of $b + K$ of the clause intervals will be scheduled. Thus, we have scheduled exactly $3a + b + K$ intervals.

We now need to show that if we can schedule $3a + b + K$ intervals, we can find a solution to the corresponding 3-Occ-Max-2SAT problem that satisfies K of the clauses. Note it is not possible to schedule more than $3a$ of the long and short intervals because for each variable, there are four overlapping intervals, and only three available machines. Suppose there exists a schedule S that schedules $3a + b + K$ intervals. We will create a modified schedule S' that schedules at least $3a + b + K$ intervals with the two following additional constraints: No long interval will be scheduled on the companion machine and, for each variable u , exactly one of its long intervals will be scheduled.

To construct S' , we first modify S so that C_1 is not used to schedule any long intervals. Suppose, in schedule S , that

machine C_1 is used to schedule one of the long intervals for some variable u . In this case, we simply take all intervals currently scheduled on the corresponding machine M_u or $M_{\bar{u}}$, schedule them on C_1 , and then schedule the long interval for u on the corresponding machine. The modified schedule completes the same number of intervals as S does.

Now, suppose, for a variable u , we use both M_u and $M_{\bar{u}}$ to schedule the corresponding long intervals. Then, at least one of the short intervals corresponding to u is not scheduled. We modify the schedule by scheduling one of the unscheduled short intervals on its corresponding machine, M_u or $M_{\bar{u}}$, and dropping the corresponding long interval. Thus, each variable u has at most one of its two long intervals scheduled on its corresponding machine, M_u or $M_{\bar{u}}$.

We now modify the schedule so that, for each variable u , exactly one of the two long intervals for u is scheduled on its corresponding machine. Let u be a variable where we do not schedule either of its corresponding long intervals. Consider the clause intervals that correspond to variable u . Because the variable u appears at most three times, either it appears in its positive form at most once or it appears in its negative form at most once. It follows that either machine M_u or machine $M_{\bar{u}}$ can schedule at most one clause interval. We then use the machine that could schedule at most one clause interval corresponding to variable u to schedule the long interval for u , dropping the single clause interval for u if necessary. If this machine was currently used to schedule the short interval for u , we modify the schedule to have C_1 schedule this short interval and the other machine, M_u or $M_{\bar{u}}$, schedule the other short interval for u .

The schedule S' now has at least $3a + b + K$ intervals scheduled since all modifications we performed did not decrease the number of intervals scheduled. For each variable, exactly one long interval will be scheduled and it will not be on C_1 . This schedule will define a valid truth assignment for 3-Occ-Max-2SAT. Specifically, if the long interval u is not scheduled, then variable u is true; otherwise, u is false. Since at most $3a$ of the short and long intervals are scheduled, at least $b + K$ clause intervals are scheduled. At most b of these clause intervals can be scheduled on C_1 , which means that at least K clause intervals will be scheduled on a machine in G . These “at least K ” clause intervals scheduled on a machine in G correspond to K satisfied clauses in the instance of 3-Occ-Max-2SAT with the above truth assignment. \square

Theorem 2.2 *The Fundamental Companion Cache Scheduling (FCCS) problem is NP-complete.*

Proof. We reduce FCIS to FCCS. For each machine in FCIS, create a cache location in D . For each interval (s_i, f_i, σ_i) , create a memory item x_i and let $g(x_i) = \sigma_i$. In other words, memory item x_i can be placed in the cache location that corresponds to the machine on which the interval (s_i, f_i, σ_i) can be scheduled. Of course, all memory items can be placed in the companion buffer C_1 as well. Each memory item will be referenced exactly twice in the memory reference sequence s . We create s as follows: Sort the start and end times $s_1, f_1, s_2, f_2, \dots, s_n, f_n$ of the intervals. Ties are broken in the following manner: End times come before start times; this essentially means that intervals that intersect only at a single point in time have no overlap. Otherwise, the start (end) time from the lower indexed interval is first; this is an arbitrary tie-breaking procedure and any such policy would suffice. This sorted list now defines the sequence of memory requests s as follows: Suppose the j th element of the sorted list is s_j or f_j . Then, memory item x_i is the j th memory item requested in s . Finally, the integer bound for FCCS is set to be Q .

Suppose Q intervals can be completed in the FCIS input instance. We now show that Q cache hits can be achieved in the corresponding FCCS input instance. For any interval j that is

not completed in the FCIS instance, we bypass both references to the corresponding memory item x_j in the FCCS instance. For any interval i that is completed in the FCIS instance, we store the first reference to the corresponding memory item x_i in the cache location that corresponds to the machine that the interval i was scheduled on. Since the interval i was completed in the FCIS instance and ties are broken with end times coming before start times, it follows that memory item x_i will still be in the cache when it is referenced the second time and a hit will be achieved. It is likewise easy to see that if Q cache hits can be achieved in the corresponding FCCS input instance, then the corresponding Q intervals can be scheduled in the FCIS input instance. Finally, it is easy to see that $FCCS \in NP$ and, thus, is NP-complete. \square

2.3 Generalizing the hardness result

The results of Section 2.2 apply only to the FCCS problem—a highly restricted instance of optimally scheduling a *CCS*. We now give proof sketches that show how Theorem 2.2 can be generalized. Full proofs are omitted due to space limitations.

Corollary 2.1. *If the main cache is set-associative, the problem is still NP-complete.*

Proof Sketch. For an n -way set-associative main cache, the difference is that we have n copies of each machine M_u and $M_{\bar{u}}$. To compensate, we create n copies of each short interval and clause interval. \square

Corollary 2.2 *If the companion buffer's size is greater than one, the problem is still NP-complete.*

Proof Sketch. We add a new machine $B \in G$ and enough new intervals that map to B to use up all the extra companion machines. \square

Corollary 2.3. *If bypassing is not allowed in the cache, the problem is still NP-complete.*

Proof Sketch. We modify the FCIS instances created by our reduction in Theorem 2.1 by breaking ties in start or end times in a consistent fashion so that the intervals are nested, that is, if two intervals intersect, then one interval must properly contain the other interval. When FCIS instances with nested intervals are reduced to FCCS instances by our construction in Theorem 2.2, they have the property that disallowing bypassing will not decrease the number of intervals which can be scheduled by an optimal algorithm. \square

Corollary 2.4. *If reordering is freely allowed in the cache, the problem is still NP-complete.*

Proof Sketch. Reordering, preemption with migration in the FCIS setting, does not allow any additional intervals to be scheduled because the structure of the resulting FCIS instance has critical periods of time where several intervals overlap with each other simultaneously. Reordering cannot increase the number of these intervals that can be scheduled. \square

Corollary 2.5. *Finding the optimal schedule for a skew cache [13] is NP-hard.*

Proof Sketch. An input instance is possible where all the accesses to one bank of the cache are to the same location. This one location is essentially the companion buffer of a *CCS*. \square

Corollary 2.6. *Optimally scheduling a multilevel cache with disjoint contents is NP-hard.*

Proof Sketch. Each set of the smaller L1 cache maps to many sets of the L2 cache. Scheduling the one set of the L1 cache and the k sets of the L2 cache is a *CCS* problem. \square

Theorem 2.3. *Unless $P = NP$, no polynomial-time algorithm with an approximation ratio better than $8,060/8,059$ can exist for FCCS.*

Proof Sketch. The reductions in Theorems 2.1 and 2.2 combine to form an approximation preserving L -reduction [12] from 3-Occ-Max-2SAT to FCCS. The actual bound in our theorem is achieved by applying our constructions directly to the 3-Occ-Max-2SAT instance in Section 4 of Berman and Karpinski's paper [3]. \square

3 CONCLUSION

We have shown that an important tool for cache studies, the optimal replacement policy, is NP-hard to compute for many nonstandard caches. While these results will not help one build faster caches, researchers who are aware of them will not waste their time in a futile attempt to find efficient algorithms for these problems. Unless $P = NP$, such algorithms do not exist.

ACKNOWLEDGMENTS

This work supported in part by US National Science Foundation grants CCR-9701679 and CCR-0105283.

REFERENCES

- [1] E.A. Arkin and E.B. Silverberg, "Scheduling Jobs with Fixed Start and End Times," *Discrete Applied Math.*, vol. 18, pp. 1-8, 1987.
- [2] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems J.*, vol. 5, no. 2, pp. 282-288, 1966.
- [3] P. Berman and M. Karpinski, "On Some Tighter Inapproximability Results," Technical Report 99-23, DIMACS, 1999.
- [4] D. Burger, J.R. Goodman, and A. Kägi, "Memory Bandwidth Limitations of Future Microprocessors," *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, pp. 78-89, 1996.
- [5] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher, and J. Zheng, "Design of the HP PA7200," *Hewlett-Packard J.*, Feb. 1996.
- [6] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*. New York: Academic Press, 1980.
- [7] A. Gonzalez, C. Aliagas, and M. Valero, "Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proc. 1995 Conf. Supercomputing*, pp. 338-347, 1995.
- [8] T.L. Johnson and W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. 24th Ann. Int'l Symp. Computer Architecture, Computer Architecture News* vol. 25, no. 2, pp. 315-326, June 1997.
- [9] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, vol. 18, no. 2, pp. 364-373, May 1990.
- [10] A.W.J. Kolen and J.G. Kroon, "On the Computation Complexity of (Maximum) Class Scheduling," *European J. Operational Research*, vol. 54, pp. 23-38, 1991.
- [11] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM System J.*, vol. 9, no. 2, pp. 78-117, 1970.
- [12] C.M. Papadimitriou, *Computational Complexity*. Addison-Wesley, 1994.
- [13] A. Seznec, "A Case for Two-Way Skewed-Associative Caches," *Proc. 20th Int'l Symp. Computer Architecture*, pp. 169-178, 1993.
- [14] R.A. Sugumar, "Multi Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs," doctoral dissertation, Univ. of Michigan, 1993.
- [15] R.A. Sugumar and S.G. Abraham, "Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization," *Proc. ACM SIGMETRICS*, pp. 24-35, May 1993.
- [16] E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, and E.S. Davidson, "Active Management of Data Caches by Exploiting Reuse Information," *IEEE Trans. Computers*, vol. 48, no. 11, Nov. 1999.
- [17] D. Truong, F. Bodin, and A. Seznec, "Accurate Data Layout into Blocks May Boost Cache Performance," *Proc. Interact-2*, pp. 55-57, Feb. 1997.
- [18] G. Tyson, M. Farrens, J. Matthews, and A.R. Pleszkun, "A Modified Approach to Data Cache Management," *Proc. 28th Ann. Int'l Symp. Microarchitecture*, pp. 93-103, 1995.
- [19] W.A. Wong and J.-L. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture*, pp. 49-60, Jan. 2000.