CSAIL

Computer Science and Artificial Intelligence Laboratory

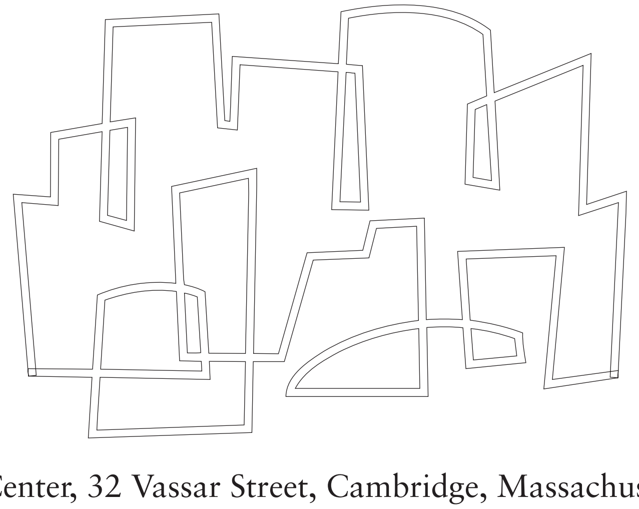
Massachusetts Institute of Technology

Software-assisted Cache Replacement Mechanisms for Embedded Systems

Prabhat Jain, Srinivas Devadas

2001, January

Computation Structures Group
Memo 435



The Stata Center, 32 Vassar Street, Cambridge, Massachusetts 02139

Software-assisted Cache Replacement Mechanisms for Embedded Systems

Prabhat Jain, Srinivas Devadas
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
e-mail: {prabhat,devadas}@mit.edu

Abstract

We address the problem of improving cache predictability (worst-case performance) and performance in embedded systems through the use of software-assisted replacement mechanisms. These mechanisms require additional software controlled state information that affects the cache replacement decision. Software instructions allow a program to kill a particular cache element, i.e., effectively make the element the least recently used element, or keep that cache element, i.e., the element will never be evicted.

We prove basic theorems that provide conditions under which kill and keep instructions can be inserted into program code, such that the resulting performance is guaranteed to be as good as or better than the original program run using the standard LRU policy. We developed algorithms based on the theoretical results that, given an arbitrary program, determine when to perform software-assisted replacement, i.e., when to insert either a kill or keep instruction. Empirical evidence is provided that shows that performance and predictability (worst-case performance) can be improved for many programs.

1 Introduction

On-chip memory, in the form of cache, scratchpad SRAM, (and more recently) embedded DRAM or some combination of the three, is ubiquitous in programmable embedded systems to support software and to provide an interface between hardware and software. Most systems have both cache and scratchpad memory on-chip since each addresses a different need. Caches are transparent to software since they are accessed through the same address space as the larger backing storage. They often improve overall software performance but are unpredictable. Although the cache replacement hardware is known, predicting its performance depends on accurately predicting past and future reference patterns. Of course, these reference patterns vary depending on input data. Scratchpad memory is addressed via an independent address space and thus must be managed explicitly by software, oftentimes a complex and cumbersome problem, but provides absolutely predictable performance. Thus, even though a pure cache system may perform better overall, scratchpad memories are necessary to guarantee that critical performance metrics are always met.

Of course, both caches and scratchpad memories should be available to embedded systems so that the appropriate memory structure can be used in each instance. A static division, however, is guaranteed to be suboptimal as different applications have different requirements. Previous research has shown that even within a single application, dynamically varying the partitioning between cache and scratchpad memory can significantly improve performance [11].

One important aspect to cache design is the choice of the replacement strategy, that controls which cache line to evict from the cache when a new line is brought in. The most commonly used replacement strategy is the Least Recently Used (LRU) replacement strategy, where the cache line that was least recently used is evicted. It is known, however, that LRU does not perform well in many situations, including timeshared systems where multiple processes use the same cache and when there is streaming data in applications. Additionally, the LRU policy often performs poorly for applications in which the cache memory requirements and memory access patterns change during execution. Furthermore, while caches improve average performance, they can cause unpredictable performance. Most cache replacement policies, including LRU, do not provide mechanisms to increase predictability (worst-case performace), making them unsuited for many real-time embedded system applications.

In this paper, we address the problem of improving cache predictability (worst-case performace) and performance through the use of software-assisted replacement mechanisms. The basic mechanism we consider is an augmentation of the least recently used (LRU) replacement method, where additional state in the cache affects the replacement decision. Software can kill a cache element, i.e., effectively make the element the least recently used element, or keep a cache element, i.e., the element will never be evicted from the cache. We consider different variations of these cache kill and keep instructions in this paper.

Our contributions are twofold. First, we provide a theoretical foundation for the development of program analysis and transformation techniques that can automatically add kill and keep instructions to a program. In particular, we prove basic theorems that provide conditions under which kill and keep instructions can be inserted into program code, such that the resulting performance, measured as the hit rate, is guaranteed to be as good as or better than the original program run using the standard LRU policy. Second, we develop algorithms based on this theory that, given an arbitrary program, determine when to perform software-assisted replacement, i.e., when to insert either a kill or keep instruction. Empirical evidence is provided that shows that performance and predictability (worst-case performace) can be improved for many programs.

The remainder of the paper is organized as follows. In Section 2, we describe related work in software-controlled caches. In Section 3, we describe our overall strategy for performance improvement. We present our theoretical results in Section 4; these provide the foundation for the trace-based and compiler-based algorithms described in Section 6. Preliminary experimental results are presented in Section 7. We provide conclusions and discuss ongoing work in Section 8.

2 Related Work

2.1 Cache Management

Some current microprocessors have cache management instructions that can flush or clean a given cache line, prefetch a line or zero out a given line [8, 10]. Other processors permit cache line locking within the cache, essentially removing those cache lines as candidates to be replaced [2, 3]. Explicit cache management mechanisms have been introduced into certain processor instruction sets, giving those processors the ability to limit pollution. One such example is the Compaq Alpha 21264 [4] where the new load/store instructions minimize pollution by invalidating the cache-line after it is used.

In [7] the use of cache line locking and release instructions is suggested based on the frequency of usage of the elements in the cache lines. In [14] some modified LRU replacement policies have been proposed to improve the second-level cache behavior that look at the temporal locality of the cache lines either in an off-line analysis or with the help of some hardware. In [12], active management of data caches by exploiting the reuse information is discussed along with the active block allocation schemes. In [5], policies in the range of Least Recently Used and Least Frequently Used are discussed.

Our work differs from previous work in that we provide hit rate guarantees when our algorithms are used to insert cache control instructions. Further, the theoretical results that we provide can be used as a basis for developing a varied set of methods for automatic cache control instruction insertion.

2.2 Memory Exploration in Embedded Systems

Cache memory issues have been studied in the context of embedded systems. McFarling presents techniques of code placement in main memory to maximize instruction cache hit ratio [9, 13]. A model for partitioning an instruction cache among multiple processes has been presented [6].

Panda, Dutt and Nicolau present techniques for partitioning on-chip memory into scratch-pad memory and cache [11]. The presented algorithm assumes a fixed amount of scratchpad memory and a fixed-size cache, identifies critical variables and assigns them to scratchpad memory. The algorithm can be run repeatedly to find the optimum performance point.

A technique to dynamically partition a cache using column caching was presented in [1]. While column caching can improve predictability for multitasking, it is less effective for single processes. Column caching requires significant cache redesign.

3 Overall Strategy

To use caches more efficiently, application-specific information should be incorporated into the cache line replacement decisions. Program analysis or trace analysis gives indications about future variable accesses that can be used to augment the LRU replacement policy with cache kill or keep instructions. These cache instructions are implemented with some additional cache replacement logic, state and tables. These instructions modify the cache replacement state and the tables. Changing the replacement state influences the replacement policy. A variety of cache control instructions with differing hardware requirements can be used.

3.1 Cache Control Instructions

We consider two forms of cache control instructions: (1) modified load/store instructions that contain the necessary cache control information (2) separate cache control instructions that contain only the cache control information. We discuss the kill, conditional kill, and keep control instructions and their hardware and software requirements.

3.2 Kill Instruction

This form of a kill instruction is a load-store instruction with the kill hint information as part of the instruction. The kill instruction allows a cache line associated with the access to be augmented with a cache line kill state. This additional kill state is used along with the LRU information to choose a cache line other than the LRU cache line for replacement. This instruction provides a mechanism for replacement of data earlier than it would be possible in the LRU policy. It can be used for references that result in the last accesses of the array elements or data structures or the references whose accessed data reuse time is such that early replacement is likely to benefit the overall performance.

3.3 Conditional Kill Instruction

This form of a kill instruction is a load-store instruction with the kill hint information as part of the instruction. The kill hint information contains the condition(s) for the cache line kill state to be updated. We consider the *offset* condition which specifies the offset(s) as a condition. A cache line kill state is updated only if an access generated by the kill instruction satisfies the offset condition. For example, consider a cache line size of 4 words $(0, ..., 3)$, an array A and a reference $A[i]$. To set the kill state of a cache line when the reference $A[i]$ results in an access to the fourth word of the cache line, the offset condition is $(A[i]\ mod\ 4) == 3$, where $(A[i]\ mod\ 4)$ specifies the offset of $A[i]$.

3.4 Keep Instruction

This form of a keep instruction is a load-store instruction with the keep hint information as part of the instruction. The keep instruction allows a cache line associated with the access to be augmented with a cache line keep state. This additional keep state provides a mechanism to keep a cache line in the cache longer than it would otherwise be kept in the cache with the LRU replacement policy. The keep state is used along with the LRU information to choose a cache line other than the LRU cache line for replacement. It can be used to keep the time-critical data in the cache for a desired period of time. We consider the use of the keep state as a *flexible keep* state that does not require a release instruction. If the keep state is used as a *fixed keep*, then a corresponding release instruction may be needed.

3.5 Hardware Cost

The use of the above instructions requires modification to the replacement logic to take into account the additional cache line states for replacement decisions. The Kill, Conditional Kill, and Keep instructions described above require only one bit of additional state per cache line in the cache. The Conditional Kill instruction requires a small amount of logic for offset matching.

3.6 Software Cost

The software cost for the above instructions is in the form of the additional flavors of load-store instructions with kill hint, keep hint, and kill with offset condition. The use of the flavors of Kill, Conditional Kill, and Keep instructions does not result in additional accesses in the instruction or data stream during program execution.

4 Theoretical Results

We present theoretical results for the replacement mechanisms that use the additional states to keep or kill the cache lines. We show the conditions under which the replacement mechanisms with the kill and keep states are guaranteed to perform better than the LRU policy.

4.1 Kill+LRU Replacement Policy

In this replacement policy, each element in the cache has an additional one-bit state (K_l) called the kill state associated with it. The K_l bit can be set under software or hardware control. On a hit the elements in the cache are reordered along with their K_l bits the same way as in an LRU policy. On a miss, instead of replacing the LRU element in the cache, an element with its K_l bit set is chosen to be replaced and the new element is placed at the most recently used position and the other elements are reordered as necessary. We consider two variations of this replacement policy to choose an element with the K_l bit set for replacement: (1) the least recent element that has its K_l bit set is chosen to be replaced; (2) the most recent element that has its K_l bit set is chosen to be replaced. The K_l bit is reset when there is a hit on an element with its K_l bit set unless the current access sets the K_l bit. The K_l bit of an element may be set even if an access is not to that element. But, we assume that the K_l bit is actually changed – set or reset – for an element only upon an access to that element. The access to the element has an associated hint that determines the K_l bit after the access and the access does not affect the K_l bit of other elements in the cache. We show the proof for variation (1) and the proof for variation (2) is similar.

Definitions: For a fully-associative cache C with associativity m, the cache state is an ordered set of elements. Let the elements in the cache have a position number in the range $[1, ..., m]$ that indicates the position of the element in the cache. Let $pos(e)$, $1 \leq pos(e) \leq m$ indicate the position of the element e in the ordered set. If $pos(e) = 1$, the e is the most recently used element in the cache. If $pos(e) = m$, then the element e is the least recently used element in the cache. Let $C(LRU, t)$ indicate the cache state C at time t when using the LRU replacement policy. Let $C(KIL, t)$ indicate the cache state C at time t when using the Kill+LRU policy. We assume the Kill+LRU policy variation (1) in the following description. Let X and Y be sets of elements and let X_0 and Y_0 indicate the subsets of X and Y respectively with K_l bit reset. Let the relation $X \preceq_0 Y$ indicate that the $X_0 \subseteq Y_0$ and the order of common elements $(X_0 \cap Y_0)$ in X_0 and Y_0 is the same. Let X_1 and Y_1 indicate the subsets of X and Y respectively with K_l bit set. Let the relation $X \preceq_1 Y$ indicate that the $X_1 \subseteq Y_1$ and the order of common elements $(X_1 \cap Y_1)$ in X_1 and Y_1 is the same. Let d indicate the number of distinct elements between the access of an element e at time t_1 and the next access of the element e at time t_2.

Lemma 1: If the condition $d \geq m$ is satisfied, then the access of e at t_2 would result in a miss in the LRU policy.
Proof of Lemma 1: On every access to a distinct element, the element e moves by one position towards the LRU position m. So, after $m-1$ distinct element accesses, the element e reaches the LRU position m. At this time, the next distinct element access replaces e. Since $d \geq m$, the element e is replaced before its next access, therefore the access of e at time t_2 would result in a miss.

Lemma 2: The set of elements with K_l bit set in $C(KIL, t) \preceq_1 C(LRU, t)$ at any time t.
Proof of Lemma 2:
At $t = 0$, $C(KIL, 0) \preceq_1 C(LRU, 0)$.
Assume that at time t, $C(KIL, t) \preceq_1 C(LRU, t)$.
At time $t + 1$, let the element that is accessed be e.
Case H: The element e results in a hit in $C(KIL, t)$. If the K_l bit for e is set, then e is also an element of $C(LRU, t)$ from the assumption at time t. Now the K_l bit of e would be reset unless it is set by this access. Thus, we have $C(KIL, t + 1) \preceq_1 C(LRU, t + 1)$. If the K_l bit of e is 0, then there is no change in the order of elements with the K_l bit set. So, we have $C(KIL, t + 1) \preceq_1 C(LRU, t + 1)$.
Case M: The element e results in a miss in $C(KIL, t)$. Let y be the least recent element with the K_l bit set in $C(KIL, t)$. If e results in a miss in $C(LRU, t)$, Let $C(KIL, t) = \{M_1, y, M_2\}$ and $C(LRU, t) = \{L, x\}$. M_2 has no element with K_l bit set. If the K_l bit of x is 0, $\{M_1, y\} \preceq_1 \{L\}$ implies $C(KIL, t + 1) \preceq_1 C(LRU, t + 1)$. If the K_l bit of x is set and $x = y$ then $\{M_1\} \preceq_1 \{L\}$ and that implies $C(KIL, t + 1) \preceq_1 C(LRU, t + 1)$. If the K_l bit of x is set and $x \neq y$, then $x \notin M_1$ because that violates the assumption at time t and $y \in L$ from the assumption at time t and this implies $C(KIL, t + 1) \preceq_1 C(LRU, t + 1)$.

Theorem: For a fully associative cache with associativity m if the K_l bit for any element e is set upon an access at time t_1 only if the number of distinct elements d between the access at time t_1 and the next access of the element e at time t_2 is such that $d \geq m$, then the Kill+LRU policy variation (1) is as good as or better than LRU.
Proof: We consider the Kill+LRU policy variation (1) for a fully-associative cache C with associativity m. We show that $C(LRU, t) \preceq_0 C(KIL, t)$ at any time t.
At $t = 0$, $C(LRU, 0) \preceq_0 C(KIL, 0)$.
Assume that at time t, $C(LRU, t) \preceq_0 C(KIL, t)$.
At time $t + 1$, let the element accessed is e.

Case 0: The element e results in a hit in $C(LRU,t)$. From the assumption at time t, e results in a hit in $C(KIL,t)$ too. Let $C(LRU,t) = \{L_1, e, L_2\}$ and $C(KIL,t) = \{M_1, e, M_2\}$. From the assumption at time t, $L_1 \preceq_0 M_1$ and $L_2 \preceq_0 M_2$. From the definition of LRU and Kill+LRU replacement, $C(LRU, t+1) = \{e, L_1, L_2\}$ and $C(KIL, t+1) = \{e, M_1, M_2\}$. Since $\{L_1, L_2\} \preceq_0 \{M_1, M_2\}$, $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.
Case 1: The element e results in a miss in $C(LRU,t)$, but a hit in $C(KIL,t)$. Let $C(LRU,t) = \{L, x\}$ and $C(KIL,t) = \{M_1, e, M_2\}$. From the assumption at time t, $\{L, x\} \preceq_0 \{M_1, e, M_2\}$ and it implies that $\{L\} \preceq_0 \{M_1, e, M_2\}$. Since $e \notin L$, we have $\{L\} \preceq_0 \{M_1, M_2\}$. From the definition of LRU and Kill+LRU replacement, $C(LRU, t+1) = \{e, L\}$ and $C(KIL, t+1) = \{e, M_1, M_2\}$. Since $L \preceq_0 \{M_1, M_2\}$, $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.
Case 2: The element e results in a miss in $C(LRU,t)$ and a miss in $C(KIL,t)$ and there is no element with K_l bit set in $C(KIL,t)$. Let $C(LRU,t) = \{L, x\}$ and $C(KIL,t) = \{M, y\}$. From the assumption at time t, there are two possibilities: (a) $\{L, x\} \preceq_0 M$, or (b) $L \preceq_0 M$ and $x = y$. From the definition of LRU and Kill+LRU replacement, $C(LRU, t+1) = \{e, L\}$ and $C(KIL, t+1) = \{e, M\}$. Since $L \preceq_0 M$ for both sub-cases (a) and (b), we have $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.
Case 3: The element e results in a miss in $C(LRU,t)$ and a miss in $C(KIL,t)$ and there is at least one element with the K_l bit set in $C(KIL,t)$. There are two sub-cases (a) there is an element with the K_l bit set in the LRU position, (b) there is no element with the K_l bit set in the LRU position. For sub-case (a), the argument is the same as in Case 2. For sub-case (b), let the LRU element with the K_l bit set is in position i, $1 \leq i < m$. Let $C(LRU,t) = \{L, x\}$ and $C(KIL,t) = \{M_1, y, M_2\}$, $M_2 \neq \phi$. From the assumption at time t, $\{L, x\} \preceq_0 \{M_1, y, M_2\}$, which implies $\{L\} \preceq_0 \{M_1, y, M_2\}$. Since y has the K_l bit set, $y \in L$ using Lemma 2. Let $\{L\} = \{L_1, y, L_2\}$. So, $\{L_1\} \preceq_0 \{M_1\}$ and $\{L_2\} \preceq_0 \{M_2\}$. Using Lemma 1, for the LRU policy y would be evicted from the cache before the next access of y. The next access of y would result in a miss using the LRU policy. So, $\{L_1, y, L_2\} \preceq_0 \{M_1, M_2\}$ when considering the elements that do not have have the K_l bit set. From the definition of LRU and Kill+LRU replacement, $C(LRU, t+1) = \{e, L_1, y, L_2\}$ and $C(KIL, t+1) = \{e, M_1, M_2\}$. Using the result at time t, we have $C(LRU, t+1) \preceq_0 C(KIL, t+1)$.

4.2 Kill+Keep+LRU Replacement Policy

In this replacement policy, each element in the cache has two additional states associated with it. One is called a kill state represented by a K_l bit and the other is called a keep state represented by a K_p bit. The K_l and K_p bits cannot both be 1 for any element in the cache at any time. The K_l and K_p bits can be set under software or hardware control. On a hit the elements in the cache are reordered along with their K_l and K_p bits the same way as in an LRU policy. On a miss, if there is an element with the K_p bit set at the LRU position, then instead of replacing this LRU element in the cache, the most recent element with the K_l bit set is chosen to be replaced by the element at the LRU position (to give the element with the K_p bit set the most number of accesses before it reaches the LRU position again) and all the elements are moved to bring the new element at the most recently used position. On a miss, if the K_p bit is 0 for the element at the LRU position, then the elements in the cache are reordered along with their K_l and K_p bits in the same way as in an LRU policy. There are two variations of this policy: (a) *Flexible Keep:* On a miss, if there is an element at the

LRU position with the K_p bit set and if there is no element with the K_l bit set, then replace the LRU element (b) *Fixed Keep:* On a miss, if there is an element at the LRU position with the K_p bit set and if there is no element with the K_l bit set, then replace the least recent element with its K_p bit equal to 0.

Theorem: Whenever there is an element at the LRU position with the K_p bit set there is also a different element with the K_l bit set, then the *Fixed Keep* variation of the Kill+Keep+LRU policy is as good as or better than LRU. Similarly, *Flexible Keep* variation of the Kill+Keep+LRU policy is as good as or better than the LRU policy.
Proof: We just give a sketch of the proof here, since the cases are similar to the ones in the Kill+LRU Theorem. We assume a fully-associative cache C with associativity m. Let $C(KKL, t)$ indicate the cache state C at time t when using the Kill+Keep+LRU policy. A different case is where the current access of an element e results in a miss in $C(KKL, t)$ and the element x at the LRU position has its K_p bit set. Consider the *Flexible Keep* variation of the Kill+Keep+LRU. If there is no element with the K_l bit set, then the element x is replaced and the case is similar to the Kill+LRU policy. If there is at least one element with the K_l bit set in $C(KKL, t)$. Let the most recent element with its K_l bit set is y. Let the cache state $C(KKL, t) = \{L_1, y, L_2, x\}$. The new state is $C(KKL, t+1) = \{e, L_1, x, L_2\}$. There is no change in the order of the elements in L_1 and L_2, so the relationship $C(LRU, t+1) \preceq_0 C(KKL, t+1)$ holds for the induction step. For the *Fixed Keep*, if there is no element with the K_l bit set, then the next more recently used element z than the element x with the K_p bit 0 is replaced. This may result in more misses for the Kill+Keep+LRU policy depending on when the element z is accessed next. So, for the *Fixed Keep* the condition in the theorem should be satisfied for it to be guaranteed to be as good as or better than the LRU policy.

5 Set-Associative Caches

SA_Kill+LRU Theorem: For a set-associative cache with associativity m if the K_l bit for any element e mapping to a cache-set i is set upon an access at time t_1 only if the number of distinct elements d mapping to the same cache-set i as e between the access at time t_1 and the next access of the element e at time t_2 is such that $d \geq m$, then the Kill+LRU policy variation (1) is as good as or better than LRU.
SA_Kill+Keep+LRU Theorem: Whenever there is an element at the LRU position with the K_p bit set in a cache-set i, there is a different element with the K_l bit set in the cache-set i, then the *Fixed Keep* variation of the Kill+Keep+LRU policy is as good as or better than LRU. Similarly, *Flexible Keep* variation of the Kill+Keep+LRU policy is as good as or better than the LRU policy.

6 Algorithm

The theoretical results presented in the previous section can be directly used if we can determine or estimate the number of distinct memory references d between two given accesses to a variable or data structure. For any pair of accesses, we do not, necessarily, have to determine d precisely, but rather we need to determine if d is less than m, where m is the associativity of the cache. We will define d to be ∞ for the last access to a variable.

We can use two basic strategies to determine if $d \geq m$, a trace-based profiling strategy or a compiler-based static analysis strategy. In order to guarantee peformance better than LRU, a lower bound on d can be used, in the cases where d is hard to estimate due to conditionals in the program or other reasons.

Either strategy has to produce the kill and keep hints that can be incorporated into the program source code or used during the compilation phase to insert appropriate kill and keep instructions into the generated code. In the sequel a reference corresponds to the source code expression or a load/store instruction that may result in different accesses. For example, given an array A, the expression $A[i]$ in the program is a reference that would generate different accesses to A depending on the value of i.

6.1 Trace-Based Algorithm

The trace-based method will work with a trace (or traces) of the program obtained using one (or more) representative input(s), and the kill and keep information obtained using the representative trace is used for other data as well. This approach works well when the control flow of the program or application does not have a significant dependence on input data.
Input:

- Cache information: Cache size, Line size, and Associativity
- A trace of the program with a representative data input
- A set of address ranges for kill and keep – these ranges typically correspond to array variables

Output:

- Hints to annotate the program with kill and keep commands or instructions

Algorithm:

1. Compute the kill hint data from the program trace
2. Map the kill hints to specific references in the program
3. Identify a reference and associate a kill command with the reference
4. Identify keep references
5. Generate the kill and keep hints for the above identified references

Computing Kill hints: The program trace is run on the given cache organization with the LRU replacement policy to derive the kill hint data. The kill hint is based on the condition $d \geq m$ outlined in the previous section. For every cache line, a time stamp, last offset access, and the reference instruction address information is maintained. On a miss the LRU cache line being replaced is checked to see if it falls within a range of the kill address ranges. If it falls within a kill address range, then a kill hint is counted for the matching kill address range and this information is maintained for mapping the kill hints to references.

Mapping Kill hints to References: The kill hints generated in the previous step are incorporated into the program trace to generate an annotated trace with kill hints for analysis. Using the kill hints in the annotated trace, the associated instruction address, and the kill address ranges information, we generate the kill hint counts for distinct reference instruction addresses.

Identifying References for Kill: We consider the kill hint count and the number of accesses generated by a reference to identify a reference for a kill instruction. For a reference, if the kill hint count is equal to the number of accesses generated, then that reference is identified as a kill reference. Otherwise, we identify the reference as a conditional kill or no-kill reference. For a conditional kill reference, we choose the offset(s) as the condition. We look at the number of accesses and the kill hints associated with each offset of the cache line in the accesses generated by a reference. If the number of accesses associated with an offset value is equal to the kill hints associated with that offset value, then the offset value forms part of the condition for a conditional kill. If no such offset is found for a reference, that reference is a no-kill reference. For a kill reference, all the accesses generated from that reference would set the K_l bit of the corresponding cache line and for a conditional kill reference, the accesses that match the offset condition would set the K_l bit of the corresponding cache line. For all the accesses generated from a no-kill reference, the K_l bit is reset for the corresponding cache line.

Identify Keep References: In this algorithm we assume that critical variables in different parts of the program are determined based on a separate analysis for predictability (worst-case performance) improvement. The references to these variables are the keep references. For a reference that is identified as a keep reference, all the accesses generated from that reference would set the K_p bit of the corresponding cache line.

Generate Hints for References: Once the references are identified as kill, conditional kill, no-kill, or keep reference, the appropriate hints are generated for these references that can be incorporated into the source code or used by the compiler during the code generation process to generate the appropriate instructions.

6.2 Compiler Algorithm

The compiler algorithm peforms many of the steps of the trace-based algorithm, however, static analysis is performed on an intermediate representation.
Algorithm:

1. Perform life-time analysis on variables in the program.

2. Choose variables as kill candidates that have a relatively short life-time or variables that are accessed infrequently, and variables as keep candidates that are referenced

many times and have a long life-time.

3. Determine a lower bound on d between each adjacent pair of references of kill variables and keep variables. For a fully associative cache, the lower bound on d is simply the minimum number of distinct references along any (control-flow) path from the first reference to the second. In the case of a set-associative cache, we require information about what cache set each reference is mapped to. The layout of variables assumed by the compiler along with the sizes of the variables accessed along each (control-flow) path is used to determine a lower bound on d. This information can be augmented by information collected from a trace using representative data as outlined in the previous section.

4. For kill variables, if any adjacent pair of references has $d \geq m$, associate a kill command with the reference. For keep variables, kill commands are generated after the last access of these variables ($d = \infty$).

5. Identify keep references. In order to guarantee performance as good or better than LRU, check to see that at each point the program, the number of killed references in the cache is equal to or greater than the number of keep references for the fixed keep method. We do not need to check this for the flexible keep method. Amongst the keep variables, the variables that are referenced more often are given higher priority for selection.

6. Generate the kill and keep hints for the above identified references

7 Preliminary Experimental Results and Analysis

We describe our experiments using the Spec95 Swim benchmark as an example. We chose a set of arrays as candidates for the kill and keep related experiments. The arrays we considered were *u, v, p, unew, vnew, pnew, uold, vold, pold, cu, cv, z, h, psi.* These variables constitute 29.1535% of the total accesses. We did the experiment with different associativity of $2, 4, 8$ and cache line size of $2, 4, 8$ words and a cache size 16K bytes. The results are shown in Figure 1. In Figure 1, the column *a,b* indicates the associativity and the cache line size in words. The column labeled *LRU* shows the hit rate over all accesses (not just the array accesses) with the LRU policy. The column labeled *Kill* shows the hit rate for the Kill+LRU replacement policy. The columns labeled *KK1, KK2, KK3* show the hit rate for the Kill+Keep+LRU replacement policy with the *Flexible Keep* variation. In *KK1*, the array variables *unew, vnew, pnew* are chosen as the keep candidates. In *KK2*, the array variables *uold, vold, pold* are chosen as the keep candidates. In *KK3*, only the array variable *unew* is chosen as the keep candidate. The hit rates of the variables of interest for the associativity 4 and cache line size 8 are shown in Figure 2 for the same columns as described above. For this example, Kill Range and Keep Range instructions were not used implying that the modified program with cache control instructions does not have any more instruction or data accesses than the original program. In Figure 2, the columns *%Imprv* show the percentage improvement in hit rate for the variables over the LRU policy. Figure 3 shows the number of

a,b	LRU	Kill	KK1	KK2	KK3
2,8	91.9342	93.6036	93.3598	92.5591	93.6505
2,4	95.2765	95.6255	95.6055	95.5335	95.6095
2,2	93.2034	93.3512	93.3265	93.2887	93.3439
4,8	91.1608	93.3711	93.8676	92.8791	93.5276
4,4	95.1871	95.6628	95.4459	95.5159	95.6368
4,2	93.0739	93.5748	93.5459	93.3225	93.5546
8,8	94.7166	96.1081	96.1470	96.0153	96.1037
8,4	95.5958	96.0604	95.9835	95.9373	96.0471
8,2	92.7306	93.2293	93.1855	93.1365	93.2134

Figure 1: Overall Hit Rates for the Spec95 Swim Benchmark

Kill (labeled as #Kill) and Conditional Kill (labeled as #Cond Kill) instructions generated corresponding the number of references (labeled as #Ref) for the array variables of the Spec95 Swim benchmark.

The results show that the performance improves significantly in some cases with the use of our software-assisted replacement mechanisms that use kill and keep instructions. The results in Figure 2 show that the hit rates associated with particular variables can be improved very significantly using our method. The bold numbers in the `KK1, KK2,` and `KK3` columns in Figure 2 indicate the hit rate of the variables that were keep variables for these columns. Choosing a particular variable and applying our method can result in an substantial improvement in hit rate and therefore performance for the code fragments where the variable is accessed. This is particularly relevant when we need to meet real-time deadlines in embedded processor systems across code fragments, rather than optimizing performance across the entire program.

Figure 4 shows the overall hit rate and performance for some Spec95 benchmarks. The columns show hit rate and number of cycles assuming 10 cycles for off-chip memory access and 1 cycle for on-chip memory access. The last column shows the performance improvement of Kill+Keep over LRU. Figure 5 shows the overall worst-case hit rate and performance for some Spec95 benchmarks. The worst-case hit rate is measured over a range of input data. The columns show hit rate and number of cycles assuming 10 cycles for off-chip memory access and 1 cycle for on-chip memory access. The last column shows the performance improvement of Kill+Keep over LRU. Results for a large set of benchmarks will be generated after our approach is fully automated.

8 Conclusions and Ongoing Work

The main contributions of our work are in laying theoretical groundwork for the development of techniques for inserting cache control instructions into programs, and the development of an algorithmic trace analysis method to automatically insert cache control instructions for improved performance. This method guarantees that the performance of the modified

Vars	LRU	Kill	%Imprv	KK1	%Imprv	KK2	%Imprv	KK3	%Imprv
u	85.9394	88.5739	3.06	86.4731	0.62	86.3232	0.44	86.6679	0.84
v	83.1306	88.1344	6.01	84.3722	1.49	84.1984	1.28	86.8634	4.49
p	84.2415	87.9287	4.37	85.4189	1.39	84.2730	0.03	86.7513	2.97
unew	28.6673	39.7238	38.56	**74.1575**	158.68	28.8033	0.47	**84.7748**	195.71
vnew	37.3741	47.1709	26.21	**75.8599**	102.97	42.8253	14.58	43.9683	17.64
pnew	31.1457	55.5108	78.22	**75.2752**	141.68	47.3057	51.88	48.2284	54.84
uold	42.6176	50.7793	19.15	47.5922	11.67	**67.7411**	58.95	47.5898	11.66
vold	54.9180	62.6039	13.99	62.4449	13.70	**75.0524**	36.66	62.4834	13.77
pold	47.3465	59.2494	25.13	55.5539	6.41	**71.4052**	33.56	55.7057	6.65
cu	75.8065	79.7312	5.54	79.3548	5.11	77.9570	3.19	79.7312	5.54
cv	75.7527	82.1505	10.28	82.1505	10.21	81.4516	7.58	82.1505	10.28
z	73.8131	84.5697	14.85	84.4955	14.85	78.2641	6.12	84.5697	14.85
h	74.1832	85.5334	18.38	85.5334	18.38	83.1595	12.28	85.5334	18.38
psi	92.3839	92.8408	0.49	92.8408	0.49	92.8408	0.49	92.8408	0.49

Figure 2: Hit Rates for the array variables in the Spec95 Swim Benchmark. The bold numbers in the `KK1`, `KK2`, and `KK3` columns indicate the hit rate of the keep variables for these columns.

Vars	#Ref	#Kill	#Cond Kill
u	28	6	10
v	28	5	12
p	24	2	11
unew	13	6	6
vnew	13	4	8
pnew	13	4	8
uold	13	5	7
vold	13	5	7
pold	13	5	7
cu	15	4	7
cv	15	2	10
z	13	5	5
h	13	4	5
psi	5	0	2

Figure 3: Number of Kill Instructions for the array variables in the Spec95 Swim Benchmark

Bench- mark	LRU Hit %	LRU Cycles	KK Hit %	KK Cycles	% Imprv Cycles
tomcatv	94.05	1105082720	95.35	1014793630	8.17
applu	97.88	113204089	97.89	113171529	0.02
swim	91.16	12795379	93.52	11188059	12.57
mswim	95.01	10627767	96.30	9715267	8.59

Figure 4: Overall Hit Rates and Performance for benchmarks. We assume off-chip memory access requires 10 processor clock cycles, as compared to a single cycle to access the on-chip cache.

Bench- mark	LRU Hit %	LRU Cycles	KK Hit %	KK Cycles	% Imprv Cycles
tomcatv	93.96	122920088	95.17	113604798	7.58
applu	97.80	635339200	97.80	635200200	0.02
swim	90.82	100394210	93.19	88016560	12.33
mswim	94.92	81969394	96.15	75290924	8.15

Figure 5: Overall Worst Case Hit Rates and Performance for benchmarks. We assume off-chip memory access requires 10 processor clock cycles, as compared to a single cycle to access the on-chip cache.

program is at least as good as the performance of the original program under the LRU replacement policy, when performance is measured in terms of hit rate. While the method described is based on the trace of the program, this method can be altered to work with information derived from an analysis of the program during program compilation. The use of cache control instructions improves the performance of a program when executed using a cache, and it improves the predictability of the program by improving its worst-case performance over a range of input data. The increased predictability afforded by cache control instructions makes caches more amenable to use in real-time embedded systems. Our preliminary experiments show that significant improvements are possible using our technique. Many different variants of this technique are possible, and we are currently exploring these variants.

References

[1] D. Chiou, S. Devadas, P. Jain, and L. Rudolph. Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[2] Cyrix. Cyrix 6X86MX Processor. May 1998.

[3] Cyrix. Cyrix MII Databook. Feb 1999.

[4] R. Kessler. The Alpha 21264 Microprocessor: Out-Of-Order Execution at 600 Mhz. In *Hot Chips 10*, August 1998.

[5] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the international conference on Measurement and modeling of computer systems*, 1999.

[6] Y. Li and W. Wolf. A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 153–156, June 1997.

[7] N. Maki, K. Hoson, and A Ishida. A Data-Replace-Controlled Cache Memory System and its Performance Evaluations. In *TENCON 99. Proceedings of the IEEE Region 10 Conference*, 1999.

[8] C. May, E. Silha, R. Simpson, H. Warren, and editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann Publishers, Inc., 1994.

[9] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the* 3^{rd} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[10] Sun Microsystems. UltraSparc User's Manual. July 1997.

[11] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Kluwer Academic Publishers, 1999.

[12] Edward S. Tam, Jude A. Rivers, Vijayalakshmi Srinivasan, Gary S. Tyson, and Edward S. Davidson. UltraSparc User's Manual. *IEEE Transactions on Computers*, 48(11):1244–1259, November 1999.

[13] H. Tomiyama and H. Yasuura. Code Placement Techniques for Cache Miss Rate Reduction. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):410–429, October 1997.

[14] W. A. Wong and J.-L Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 1999.