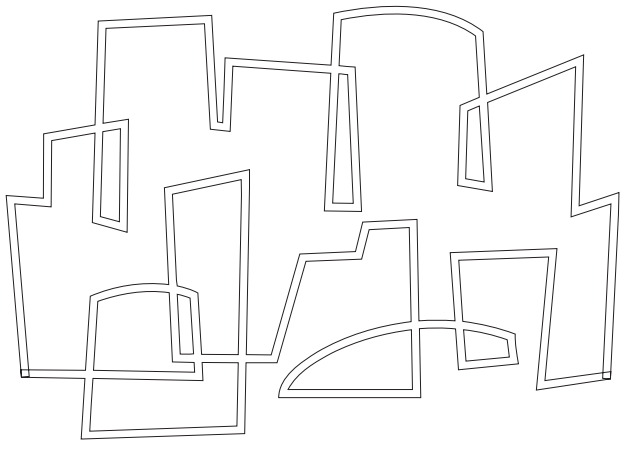
Controlling Cache Pollution in Prefetching With Software-assisted Cache Replacement

Prabhat Jain, Srini Devadas, Larry Rudolph

2001, July

Computation Structures Group
Memo 462



Controlling Cache Pollution in Prefetching With Software-assisted Cache Replacement

Prabhat Jain
prabhat@lcs.mit.edu

Srinivas Devadas
devadas@mit.edu

Larry Rudolph
rudolph@lcs.mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Aggressive prefetch methods can suffer from cache pollution when prefetched data replaces useful data in the cache, causing performance degradation. In this paper, we present a methodology that ensures that cache pollution does not degrade overall performance when software or hardware prefetching methods are used.

Software instructions can allow a program to kill a particular cache element, i.e., effectively make the element the least recently used element. We provide conditions under which kill 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. Using these results, it is possible to analyze code and determine when to perform software-assisted replacement, i.e., when to insert a kill instruction. The result of this analysis is a modified cache replacement method that may be used independently to improve cache performance, or may be used to control cache pollution caused by arbitrary prefetching methods.

A prefetching method can be combined with this modified cache replacement method in different ways, by distinguishing normal data from prefetched data. Different ways of combining prefetching with replacement have different associated performance guarantees.

We consider aggressive, sequential prefetching methods which prefetch multiple cache blocks on a cache miss. These methods are easy to implement in hardware, but may cause significant cache pollution, and/or require increased bandwidth into the cache, which may result in worse performance than not prefetching at all. We show that both bandwidth and pollution can be controlled effectively using our software-assisted replacement algorithm. Empirical evidence is provided that shows that cache performance can be significantly improved, and minimum-performance guarantees provided, using a combination of simple, aggressive hardware prefetching and software-controlled replacement.

1 Introduction

Caches are used to improve the average performance of application-specific and general-purpose processors. Caches vary in their organization, size and architecture. Depending on the cache characteristics, application performance may vary dramatically. Caches are valuable resources that have to be managed properly in order to ensure the best possible program performance.

The performance of a cache can be measured in terms of its hit or miss rate. A miss in a cache can be either a cold miss, a conflict miss, or a capacity miss. Conflict and capacity misses can sometimes be avoided using different data mapping techniques or increasing the associativity of the cache. Cold misses can be avoided using prefetching, i.e., predicting that the processor will make a request for a data block, and bringing the data in before it is accessed by the processor.

Prefetching methods come in many different flavors, and to be effective they must be implemented in such a way that they are timely, useful, and introduce little overhead [23]. In this paper, we are primarily concerned with *cache pollution* due to prefetching, where a prematurely

prefetched block displaces data in the cache that is in use or will be used in the near future by the processor [3]. Cache pollution can become significant, and cause severe performance degradation when the prefetching method used is too aggressive, i.e., too much data is brought into the cache, or too little of the data brought into the cache is useful. It has been noted that hardware-based prefetching often generates more unnecessary prefetches than software prefetching [23]. Many different hardware-based prefetch methods have been proposed (cf. Section 6); the simple schemes usually generate a large number of prefetches, and the more complex schemes usually require hardware tables of large sizes.

In this paper, we present a methodology which ensures that cache pollution does not degrade overall performance when any software or hardware prefetching method is used. Our methodology is based on combining prefetch methods with software-assisted cache replacement methods. We distinguish between normal data brought in on a cache miss from prefetched data and make different replacement decisions for each type. This allows us to effectively control the cache pollution caused by a prefetch method which in turn allows us to make performance guarantees, measured in terms of miss rate.

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. Software instructions can augment LRU replacement by allowing a program to kill a particular cache element, i.e., effectively making the element the least recently used element, or keeping that cache element, i.e., the element will never be evicted.

We provide conditions under which kill 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. Compiler-based analysis can be used, given an arbitrary program, to determine when to perform software-assisted replacement, i.e., when to insert a kill instruction. The results of this analysis can be used independently to improve cache performance, or can be used to control cache pollution caused by arbitrary prefetching methods.

Our results imply that, if for any period of time, there are n dead elements in the cache, then n elements can be kept in the cache for that period, regardless of the memory reference pattern, without loss of performance. Thus, n elements can be prefetched to replace the dead elements and performance will not degrade regardless of the usefulness of the n prefetched elements. Depending on how we treat normal data and prefetched data, any prefetching method can be combined with software-assisted replacement in different ways, each with different performance guarantees.

We consider hardware sequential prefetch methods [21] that prefetch multiple, adjacent cache blocks on a cache miss. These methods are easy to implement, but may cause significant cache pollution when multiple blocks are prefetched [20]. We show that pollution can be controlled effectively using our replacement algorithm. Analysis shows that certain performance guarantees can be made, and empirical evidence is provided that shows that performance is improved over the no-prefetch and LRU-based prefetch cases, using software-controlled replacement.

The remainder of the paper is organized as follows. In Section 2, we describe our overall strategy for controlling cache pollution using software-assisted replacement. We describe the different ways of combining prefetching with software-assisted replacement and discuss their performance. We present a condition to determine whether a cache element can be killed in Section 3, and comment

on performance guarantees for prefetching methods. We describe cache kill instructions and a compiler-based algorithm that determines when and where to insert kill instructions in Section 4. We also describe the hardware cost associated with the kill instructions and the required software support. In Section 5 we describe the prefetch method and the experiments we have conducted. In Section 6, we describe related work in prefetching and software-controlled caches. We provide conclusions and discuss ongoing work in Section 7.

2 Overall Strategy

We consider different ways of combining software-assisted replacement with a generic prefetching method in this section. We describe certain relationships between the miss-rates of the different combinations.

2.1 LRU Replacement with Kill

We first describe a modified LRU replacement policy that incorporates information about "useless" data in the cache.

In the Kill + LRU replacement policy, each element, i.e., cache line or cache block, in the cache has an additional one-bit state (K_l) called the kill state associated with it. In this paper, we will describe ways that a compiler can insert instructions into compiled code that set the K_l bit for particular cache elements, however, for the purposes of this section, it does not matter how these kill bits are set.

The K_l bit for an element may be set under software or hardware control if the element will not be reused before being replaced. Assume that some subset of the elements in the cache have their K_l bits set. If an element has its K_l bit set, the element is deemed to be *dead*.

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, if it exists, and the new element is placed at the most recently used position and the other elements are shifted or not shifted as in the LRU policy.

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. We will refer to these two variants as least-recent-kill (LCK) and most-recent-kill (MCK). When we refer to Kill + LRU replacement we will mean the least-recent-kill variation in the rest of this paper.

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. We describe the setting of the kill bits in Section 4.1.

2.2 Integration of Modified LRU Replacement with Prefetching

There are four different scenarios shown in Figure 1.

(a): Figure 1(a) illustrates the standard LRU method without prefetching. New data is brought into the cache on a miss, and the LRU replacement policy is used to evict data in the cache. We denote this method as (LRU, ϕ), where the first item in the 2-tuple indicates that new

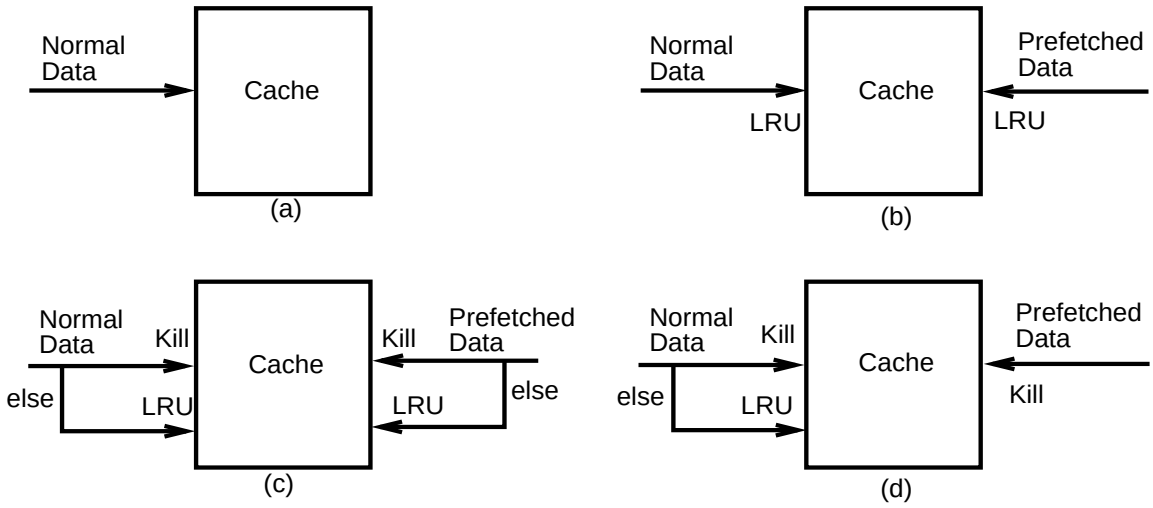


Figure 1: Integrating Prefetching with Modified LRU Replacement

(normal) data brought in on a cache miss replaces old data based on the LRU policy, and the ϕ indicates that there is no prefetched data.

(b): Figure 1(b) illustrates generic prefetching integrated with standard LRU replacement. We denote this method as (LRU, LRU) – normal data brought in on a cache miss as well as prefetched data replace old data based on the LRU policy.

(c): Figure 1(c) integrates Kill + LRU replacement with a generic prefetching method. Normal data brought in on a cache miss as well as prefetched data replaces old data based on the Kill + LRU replacement strategy described in Section 2.1. We denote this method as (Kill + LRU, Kill + LRU).

(d): Figure 1(d) integrates Kill + LRU replacement with a generic prefetching method in a different way. Normal data brought in on a cache miss replaces old data based on the Kill + LRU policy, but prefetched data is only brought in if there is enough dead data. That is, prefetched data does not replace data which does not have its kill bit set. Further, when the prefetched data replaces dead data, the LRU order of the prefetched data is *not* changed to MRU, rather it remains the same as that of the dead data. We denote this method as (Kill + LRU, Kill).

There is one other variant that is possible, the (LRU, Kill + LRU) variant. We will not consider this variant in this paper since it does not have any interesting properties.

2.3 Properties of Prefetch Methods

It should be intuitively clear, and it can be rigorously shown that

- (Kill + LRU, Kill + LRU) will perform better than (LRU, LRU) if any prefetching strategy, which is not predicated on cache hits or misses, is used, as long as it is the same in both cases, and
- (Kill + LRU, Kill) will perform better than (LRU, ϕ) for any prefetching strategy.

Note that (Kill + LRU, Kill + LRU) could actually perform worse than (LRU, ϕ).[1] Similarly, (Kill + LRU, Kill + LRU) could perform worse or better than (Kill + LRU, Kill).

3 Theoretical Results

We present theoretical results for the Kill + LRU replacement policy and prefetch methods with integrated software-assisted replacement in this section.

3.1 Kill + LRU Theorems

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 $pos(e) = 1$, then 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 sequel. 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.

We first give a simple condition that determines whether an element in the cache can be killed. This condition will serve as the basis for compiler analysis to determine variables that can be killed.

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: 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: The proof is based on induction on the cache states $C(KIL, t)$ and $C(LRU, t)$. The intuition is that after every access to the cache (hit or miss), the new cache states $C(KIL, t + 1)$ and $C(LRU, t + 1)$ maintain the relation $\preceq_1$ for the elements with the K_l bit set. Please refer to the Appendix A.1 for a detailed proof. ■

[1] Of course, if a good prefetch algorithm is used, it will usually perform better.

Theorem 1 *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: The proof is based on induction on the cache states $C(LRU, t)$ and $C(KIL, t)$. The intuition is that after every access to the cache, the new cache states $C(LRU, t+1)$ and $C(KIL, t+1)$ maintain the $\preceq_0$ relation for the elements with the K_l bit reset. In addition, the new cache states $C(KIL, t+1)$ and $C(LRU, t+1)$ maintain the relation $\preceq_1$ based on Lemma 2. Therefore, every access hit in $C(LRU, t)$ implies a hit in $C(KIL, t)$ for any time t. Please refer to Appendix A.2 for a detailed proof. ■

Theorem 1 can be generalized to set-associative caches. We omit the proof of the following theorem since it is similar to the proof for Theorem 1.

Theorem 2 *For a set-associative cache with associativity m if the K_l (kill) 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.*

3.2 Performance Guarantees for Prefetching

Using Theorems 1 and 2, we can show that the (Kill + LRU, Kill + LRU) strategy is guaranteed to be as good or better than (LRU, LRU) strategy.

Theorem 3 *Given a set-associative cache, Strategy (Kill + LRU, Kill + LRU) where Kill + LRU corresponds to variation (1) is as good as or better than Strategy (LRU, LRU) for any prefetch strategy not predicated on hits or misses in the cache.*

Proof: (Sketch) Consider a processor with a cache C_1 that runs Strategy (Kill + LRU, Kill + LRU), and a processor with a cache C_2 that runs Strategy (LRU, LRU). We have two different access streams, the normal access stream, and the prefetched access stream. Accesses in these two streams are interspersed to form a composite stream. The normal access stream is the same regardless of the replacement policy used. The prefetched access stream is also the same because even though the replacement strategy affects hits and misses in the cache, by the condition in the theorem, the prefetch requests are not affected. Therefore, in the (Kill + LRU, Kill + LRU) or the (LRU, LRU) case, accesses in composite stream encounter (Kill + LRU) replacement or LRU replacement, respectively. Since it does not matter whether the access is a normal access or a prefetch access, we can invoke Theorem 2 directly. ■

We can show that (Kill + LRU, Kill) is always as good or better than (LRU, ϕ).

Theorem 4 *Given a set-associative cache, the prefetch strategy (Kill + LRU, Kill) where the Kill + LRU policy corresponds to variation (1) is as good or better than the prefetch strategy (LRU, ϕ).*

Proof: (Sketch) The prefetch strategy (Kill + LRU, Kill) results in two access streams, the normal accesses and the prefetched blocks. The no-prefetch (LRU, ϕ) strategy has only the normal access stream. Invoking Theorem 2, the number of misses in the normal access stream is no more using Kill + LRU replacement than LRU replacement. The prefetch stream in the (Kill + LRU, Kill) strategy only affects the state of the cache by replacing a dead element with a prefetched element. When this replacement occurs, the LRU ordering of the dead item is not changed. If the element has been correctly killed, then no subsequent access after the kill will result in a hit on the dead element before the element is evicted from the cache. A dead element may result in a hit using the Kill + LRU replacement due to the presence of multiple dead blocks, but the same block would result in a miss using the LRU replacement. Therefore, we can replace the dead element with a prefetched element without loss of hits/performance. It is possible that prefetched item may get a hit before it leaves the cache, in which case performance improves. ■

4 Caches with Software-Controlled Kill

4.1 Cache Control Instructions

We consider two forms of cache control instructions: (1) modified load/store instructions that contain the necessary cache control information and (2) separate cache control instructions that contain only the cache control information. We discuss the Kill load/store and Kill range control instructions and their hardware and software requirements in this section.

4.1.1 Kill Load/Store Instruction

In Section 2.1 we described how each cache element (line or block) has an associated kill bit. The kill bit can be set by a Kill load/store instruction. A kill instruction is a load/store instruction with the kill hint information as part of the instruction. The Kill load/store instruction is a load/store instruction that loads a word into a register and sets the kill bit associated with the cache line of the word accessed. The kill bit of a cache line is reset when a non-kill load/store instruction results in a hit in the cache line that has its kill bit set. The use of this type of kill load/store instruction assumes that the compiler is cache-aware and knows the data layout as well as the block size.

4.1.2 Kill Range Instruction

This form of instruction can be used to kill parts of arrays or other data structures. It is a separate cache control instruction that specifies an address range. When an access falls within the specified address range the instruction can be used to update the cache line or word kill state that is used for the cache line replacement.

The Kill range instruction sets hardware tables that are checked each time a new block is brought into the cache. On a miss, when a block is brought in, we find the set that the block will be placed in. Before we choose the block that will be replaced within the set using the kill + LRU replacement policy, we check the Kill range hardware table to see if any of the blocks have been killed. If so, we set the kill bits in the block(s) currently in the cache, and then select the block to be replaced.

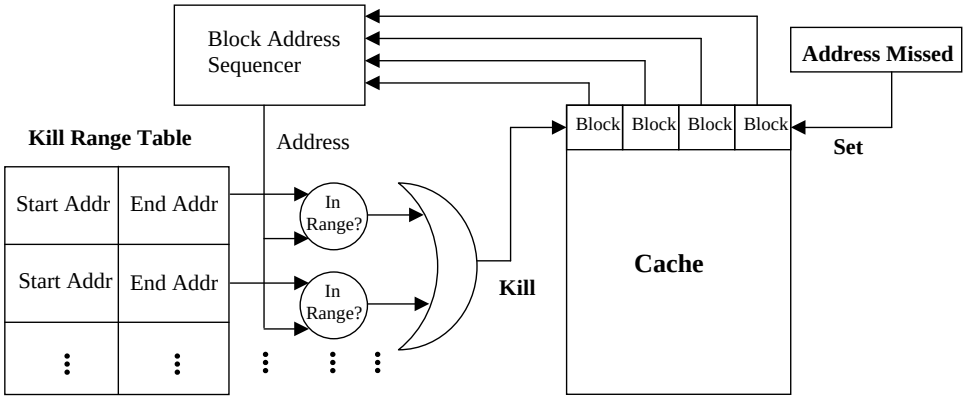


Figure 2: Kill Range Instruction Hardware

For example, consider an array $A[0, ..., 99]$ that is used in a loop with a reference $A[i]$. To set the cache kill state for the second half of the array, we can use a Kill range instruction $Kill_R(\&A[50], \&A[99])$ *before* the loop. During the execution of the second half of the loop, or after the execution of the loop, on a miss, if the set that is indexed contains $A[50-99]$ in any of its blocks, then these kill bits in these blocks (or appropriate words in the blocks) will be set prior to selecting the block within the set to be replaced.

4.1.3 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 additional logic modification is quite simple. We can find whether the block is dead or not by checking the kill bit associated with the block. For the least-recent-kill variant, the kill status of each block can be viewed as an additional most significant LRU bit, so killed blocks are always replaced first, and amongst the killed blocks the least recent block is replaced. The most-recent-kill variant is similar, once the killed blocks are identified, the most recent killed block within the set is selected. If there are no killed blocks, the least recent block is selected.

The Kill range instructions require hardware tables that store the address range information and the logic for lookup and comparison of address ranges. A Kill range hardware table consists of the Start-Address and End-Address entries. The entries of the table can be updated in the FIFO manner or using some other policy as desired. Upon an access to a set in the cache, if the address results in a miss, then the addresses corresponding to each tag in the set indexed by this address are provided to the hardware range table for comparison sequentially, and if any address falls within a range, then kill bits in the block (or kill bits in the word) corresponding to the address are set. When an address is provided for comparison, it is compared with the stored address ranges in parallel.

The above process is illustrated in Figure 2. Since the hardware table is only accessed upon a miss to this level of cache, we can perform several comparisons before data returns from the next level of the memory hierarchy and needs to be placed in this level of cache.

The maximum number of table entries for the hardware tables can be determined based on how much area is available.

4.1.4 Software Cost

The software cost for the above instructions is in the form of the additional flavors of load/store instructions with kill and an additional instruction for the Kill range instruction. The use of the Kill load/store instructions does not result in additional accesses in the instruction stream. The use of the Kill Range instructions results in additional instructions in the instruction stream. These control instructions typically represent a very small overhead in the instruction stream in our experiments.

4.2 Inserting Kill Instructions

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. For a fully associative cache, estimating d is easier, because we do not need information about what set a reference corresponds to. This information is needed for a set-associative cache. We will define d to be ∞ for the last access to a variable.

We use a compiler-based static analysis strategy. In order to guarantee performance 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 if data layout information is incomplete.

The strategy has to produce the kill hints that can be incorporated into the program source code or used during the compilation phase to insert appropriate kill 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 multiple accesses. For example, given an array A, the expression $A[i]$ in the program is a reference that would generate multiple accesses to the array if $A[i]$ is part of a loop that is iterated multiple times.

Compiler Algorithm

1. Perform life-time analysis on variables in the program, and identify the last use of all variables in the program.

2. Choose variables as kill candidates that are local to a procedure, or shared in a small number of procedures. Also choose global or local variables that have a relatively short life-time and variables that are accessed infrequently.

3. Determine a lower bound on d between each adjacent pair of references of kill candidate variables.

   In the case of a set-associative cache, we require information about what cache set and block each reference is mapped to. For statically-allocated variables, 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. For large dynamically-allocated variables such as arrays, allocated in a contiguous region of memory, parts of the array will map to all the sets of the cache. When these arrays are accessed between the accesses to a kill candidate variable, we use the size of the arrays and the number and type of accesses to determine how

many times, if any, the set corresponding to the kill candidate variable is touched by an array access. We ignore any intermediate accesses to small dynamically-allocated variables, though we might generate kills for these variables. Thus, given a pair of accesses to a kill candidate variable, we determine how many accesses to other variables fall into the same set on each control flow path. The minimum number of distinct accesses over all control flow paths is the value used for d.

For a fully associative cache, the lower bound on d is the minimum number of distinct block references along any (control-flow) path from the first reference to the second. We do not have to deal with the mapping to sets.

4. For kill candidate variables, if any adjacent pair of references has $d \geq m$, associate a kill instruction with the first reference.

5. Kill instructions are generated after the last access of all variables (since $d = \infty$). We use Kill range instructions in this case.

5 Prefetch Methods and Experiments

We first describe the hardware prefetch method we use in this section. Next, we present results integrating trace-based ideal kill with the prefetch method in various ways. Finally, we present results using compiler-inserted kills, and show that significant performance improvements can be gained while providing performance guarantees.

5.1 Hardware Prefetch Method

We consider a parameterizable variation on a sequential hardware prefetch method. We integrate this method with Kill + LRU replacement using the strategies of Figure 1(c) and 1(d).

Upon a cache miss, a block has to be brought into the cache. This block corresponds to new, normal data being brought in. Simultaneously, we prefetch i adjacent blocks and bring them into the cache, if they are not already in the cache. If i is 2, and the block address of the block being brought in is A, then we will prefetch the $A + 1$ and $A + 2$ blocks. The i^{th} block will be tagged, if it is not already in the cache. This first group of $i + 1$ blocks will either replace dead cache blocks or LRU blocks, in that order of preference.

We conditionally prefetch j more blocks corresponding to blocks with addresses $A + i + 1, \ \ldots A + i + j$, provided these blocks only replace cache blocks that are dead. This check is performed *before* prefetching.

If there is a hit on a tagged block, we remove the tag from the block, and prefetch the two groups of i and j blocks, as before. The i^{th} and last block in the first group will be tagged, if it is not already in the cache. We denote this parameterizable method as ((Kill + LRU)$-i$, (Kill)$-j$).

We now give some details pertaining to the hardware implementation of the prefetching method. We have already described how the modified replacement strategy is implemented in Section 4.1.3. Generating the new addresses for the blocks is trivial. To control the required bandwidth for prefetching, we will only prefetch the second[2] group of j blocks if these blocks will only replace

[2] This will be the second group assuming $i > 0$.

dead cache blocks. Before a prefetch request is made for a block, the cache is interrogated to see if the block is already present in the cache. This involves determining the set indexed by the address of the block to be prefetched. If this set contains the block the prefetch request is not made. Further, for the second group of j blocks, if the set does not contain at least one dead block, the prefetch request is not made. Given the set, it is easy to perform this last check, by scanning the appropriate kill bits.

The bandwidth from the next level of cache to the cache that is issuing the prefetch requests is an important consideration. There might not be enough bandwidth to support a large degree of prefetching ($i + j$ in our case). Limited bandwidth will result in prefetches either not being serviced, or the prefetched data will come in too late to be useful. Checking to see that there is dead space in the second set of j blocks reduces the required bandwidth, and ensures no cache pollution by the second set of prefetched blocks. Of course, for all prefetched blocks we check if there is dead data that can be replaced, before replacing potentially useful data, thereby reducing cache pollution.

5.2 Experiments With Ideal Kill

We present results using trace-based ideal kill, where any cache element that will not be accessed before being replaced is killed immediately after the last access to the element that precedes replacement. These results give an upper bound on the performance improvement possible using integrated kill and prefetch methods.

We compare the following replacement and prefetch strategies:

- (LRU, LRU$-i$): Standard LRU with i-block prefetching. When $i = 0$ there is no prefetching done.
- ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$): The first block and the first i prefetched blocks will replace either a dead block or an LRU block, but the next j prefetched blocks will only be brought into the cache if they are to replace dead blocks. (cf. Figure 1(c) and (d)).

We cannot prove that ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) is better than (LRU, LRU$-i$) for arbitrary i and j because Theorem 3 does not hold. For $i = 0$ and arbitrary j Theorem 4 holds. However, it is very unlikely that the former performs worse than the latter for any $i,\ j$. Kill + LRU can only result in fewer misses than LRU on normal accesses, and therefore fewer prefetch requests are made in the Kill + LRU scheme, which in turn might result in a miss for Kill + LRU and a hit for LRU later on. Of course, a future miss will result in a similar, but not necessarily identical, prefetch request from Kill + LRU. However, since the prefetch requests in the two strategies have slightly different timing and block groupings, a convoluted scenario might result in the total number of misses for ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) being more than (LRU, LRU$-i$). Our experiments indicate that this does not occur in practice.

We show experimental results for some of the integer and the floating-point benchmarks from the Spec95 (`tomcatv`) and Spec2K (`art`, `bzip`, `gcc`, `mcf`, `swim`, `twolf`, `vpr`) benchmark suites. In Figures 3 through 10, we show (on the left) the hit rates achieved by ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) and (LRU, LRU$-i$) for $0 \leq i \leq 3$ and $0 \leq i + j \leq 7$. These hit rates correspond to a 4-way set-associative cache of size 16KB with a block size of 32 bytes. We compiled the

benchmarks for the Alpha processor instruction set. We generated traces for the benchmarks using SimpleScalar 3.0 [1] and chose a 500 million total references (instruction + data) sub-trace from the middle of the generated trace. We used a hierarchical cache simulator, *hiercache*, to simulate the trace assuming a infinite-sized L2 cache. We are interested in gauging the performance of the L1 cache prefetch methods. We also placed a realistic bandwidth constraint on L2 to L1 data transfers.

In each graph, we have five curves. The first curve corresponds to LRU-x, with x varying from 0 to 7. For most benchmarks LRU-x's performance peaks at $x = 1$ or $x = 2$. The next four curves correspond to ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) for i varying from 0 to 3. The ordinate on the x-axis corresponds to the total number of prefetched blocks, i.e., $i + j$. For example, in the ART benchmark Ideal Kill graph, for $i = 3$ and $j = 4$ ($i + j = 7$) we get a 78.1% hit rate.

The benchmarks all show that increasing j for a given i does not hurt the hit rate very much because we are only prefetching blocks if there is dead space in the cache. Increasing j beyond a certain point does hurt hit rate a little because of increased bandwidth from L2 to L1 when there is a lot of dead data in the cache, but not because of cache pollution. In general, for the benchmarks ((Ideal Kill + LRU)$-i$, (Ideal Kill)$-j$) performs very well for $i = 1$ and $j \geq 0$. We conjectured that it is always (though not provably) better than LRU-i, which is borne out by the results. It is also better in all benchmarks than the no-prefetch case LRU-0.

The results show that the integration of Kill + LRU and an aggressive prefetching strategy provides hit rate improvement, significantly so in some cases. Obviously, the hit rates can be improved further, by using more sophisticated prefetching strategies, at the potential cost of additional hardware. Many processors already incorporate sequential hardware prefetch engines, and their performance can be improved using our methods.

5.3 Experiments with Compiler-Inserted Kill

We repeat the above experiments except that instead of using trace-based ideal kill we use compiler-inserted kill as described in Section 4.2.

The compiler-inserted kill performs nearly as well as ideal kill. The graphs are nearly identical in many cases. The main differences between the compiler-inserted kill method and the ideal kill method is summarized in Figure 11. In the table, the number of kill hints for the trace-based ideal kill is given first. The number of *executions* of Kill load/store and Kill range instructions is given. Note that the executed Kill Range instructions are a very small part of total instructions executed, and their performance impact is negligible. In fact, for these benchmarks, the Kill Range instructions could have been replaced by Kill load/store instructions, with no performance impact. The number of kill hints generated by the compiler is smaller, mainly because the compiler uses a conservative strategy when inserting kill instructions.

5.4 Discussion

In Figure 12, we give the hit rates corresponding to four important prefetching strategies, LRU-1, LRU-2, ((Ideal Kill + LRU)-1, (Ideal Kill)-1) shown as Ideal(1,1) in the table, and Comp(1,1), the compiler-kill version. Ideal(1,1) is always better than LRU-1, LRU-2 and the no-prefetch case. The same is true for Comp(1,1), except for ART, where Comp(1,1) is fractionally worse than

LRU-2. LRU−1 and LRU-2 are not always better than the no-prefetch LRU-0 case, for example, in SWIM and TOMCATV. The results clearly show the stability that Kill + LRU replacement gives to a prefetch method.

The memory performance numbers assuming a L2 latency of 18 cycles are given in Figure 13. Note that this is not overall performance improvement since all instructions do not access memory. The percentage of memory operations is given in the last column of the table.

Our purpose here was to show that cache pollution can be controlled by modifying cache replacement, and therefore we are mainly interested in memory performance. The effect of memory performance on overall performance can vary widely depending on the type of processor used, the number of functional units, the number of ports to the cache, and other factors. The empirical evidence supports our claim that modified cache replacement integrated with prefetching can improve memory performance.

It appears that a scheme where 1 or 2 adjacent blocks are prefetched on a cache miss, with 2 blocks being prefetched only in the case where the second block replaces a dead block in the cache improves memory performance over standard LRU-based sequential hardware prefetching. Further, this method is significantly more stable than standard LRU-based sequential hardware prefetching, which might actually hurt performance, in some cases. While we are experimenting with more sophisticated prefetching schemes that may achieve better hit rates and performance to gauge the effect of modified cache replacement, we believe that simple schemes with low hardware overheads are more likely to be adopted in next-generation processors. Our work has attempted to improve both the worst-case and average performance of prefetching schemes.

6 Related Work

6.1 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 [15, 16]. Other processors permit cache line locking within the cache, essentially removing those cache lines as candidates to be replaced [4, 5]. 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 [7] where the new load/store instructions minimize pollution by invalidating the cache-line after it is used.

In [14] 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 [24] 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 [22], active management of data caches by exploiting the reuse information is discussed along with the active block allocation schemes. In [11], 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 algorithm is used to insert cache control instructions. Further, the theoretical results that we provide have been used as a basis for developing a varied set of methods for automatic cache control instruction insertion, and can be applied to prefetching as well.

6.2 Software Prefetching

There have been many proposals for software prefetching in the literature. Many techniques are customized toward loops or other iterative constructs (e.g., [8]). These software prefetchers rely on accurate analysis of memory access patterns to detect which memory addresses are going to be subsequently referenced [17] [13] [12] [18].

Software has to time the prefetch correctly – too early a prefetch may result in loss of performance due to the replacement of useful data, and too late a prefetch may result in a miss for the prefetched data. We do not actually perform software prefetching in this paper – instead we use software control to mark cache blocks as dead as early as possible. This significantly reduces the burden on the compiler, since cycle-accurate timing is not easy to predict in a compiler.

6.3 Hardware Prefetching

Many different hardware sequential prefetching methods have been proposed, where prefetches for adjacent blocks are issued when block b is accessed. The most common approach is the one block lookahead approach (OBL) which initiates a prefetch for block $b + 1$ when block b is accessed. Smith [21] summarizes several of these approaches, of which the prefetch-on-miss and tagged prefetch algorithms are the most popular.

The OBL method can be generalized by prefetching $K > 1$ blocks, which aids the memory system in staying ahead of rapid processor requests for sequential data blocks. However, Przybylski [20] found that additional memory traffic overhead and cache pollution generated tends to make sequential prefetching infeasible for values of K larger than 1.

The difference between the above schemes and the scheme we use (cf. Section 5) is that we (may) check to see that the prefetched blocks will only replace killed cache blocks before we actually prefetch the blocks. This alleviates both the increased bandwidth required when prefetching multiple blocks, and mitigates, if not eliminates, the cache pollution problem.

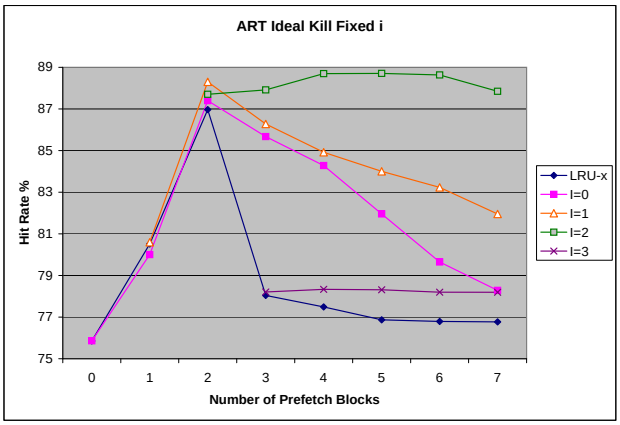
The work of [9, 10] was one of the first to combine dead block predictors and prefetching. The prefetching method is intimately tied to the determination of dead blocks. Here, we have decoupled the determination of dead blocks from prefetching – any prefetch method can be controlled using the notion of dead blocks. We have used compiler analysis to determine dead variables/blocks, and a simple hardware prefetch technique that requires minimal hardware support. The predictor-correlator methods in [10] achieve significant speedups, but at the cost of large hardware tables that require up-to 2MB of storage.

Prefetching and replacement decisions have been made in tandem in the context for I/O prefetching for file systems [2] [19].

6.4 Integrated Software and Hardware Prefetching

Integrated techniques are designed to take advantage of compile-time program information without introducing unnecessary instruction overhead as in pure software prefetching.

Variations on tagged prefetching in which the degree of prefetching K for a particular reference stream is determined at compile time and passed onto hardware have been proposed [6]. Zhang and Torellas [25] tag memory locations using a compiler such that a reference to a tagged memory location results in the prefetching of another.



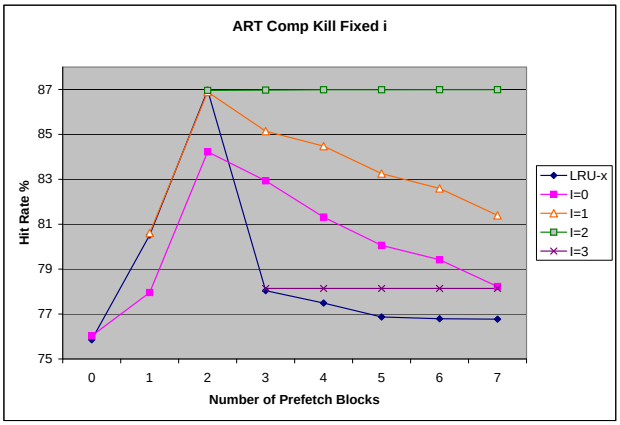
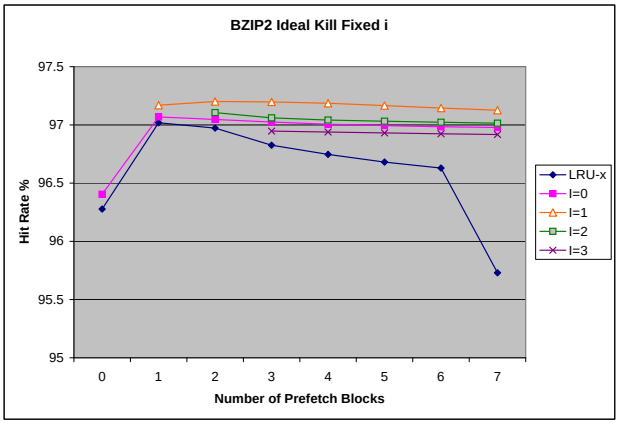


Figure 3: ART L1 Hit Rate



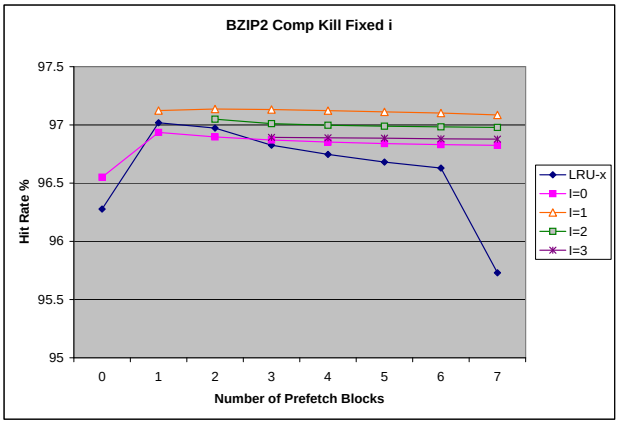
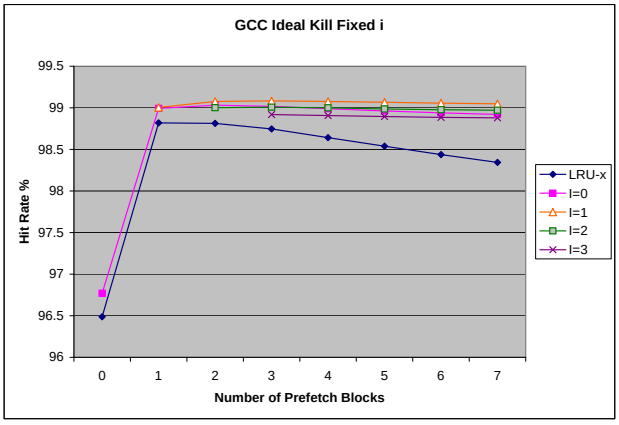


Figure 4: BZIP L1 Hit Rate



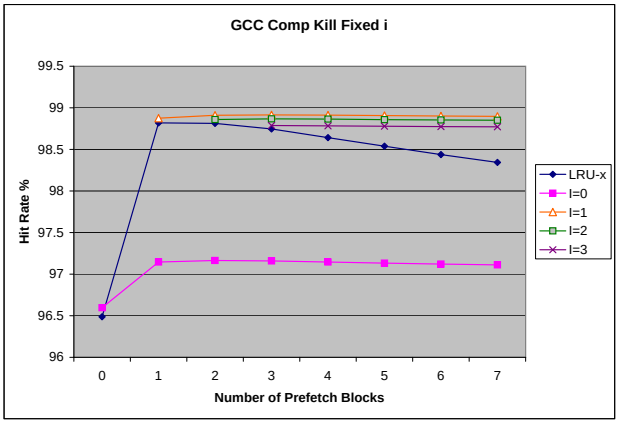


Figure 5: GCC L1 Hit Rate

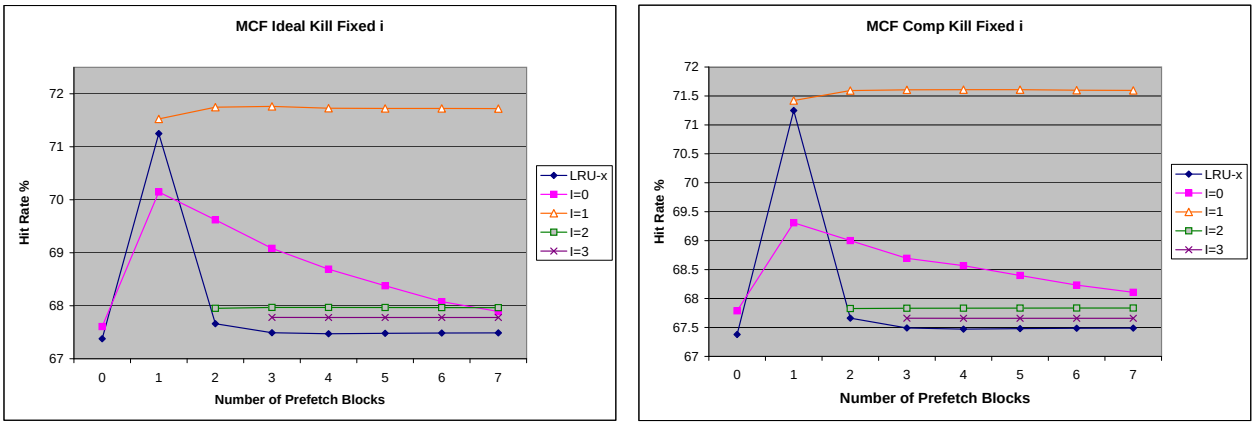


Figure 6: MCF L1 Hit Rate

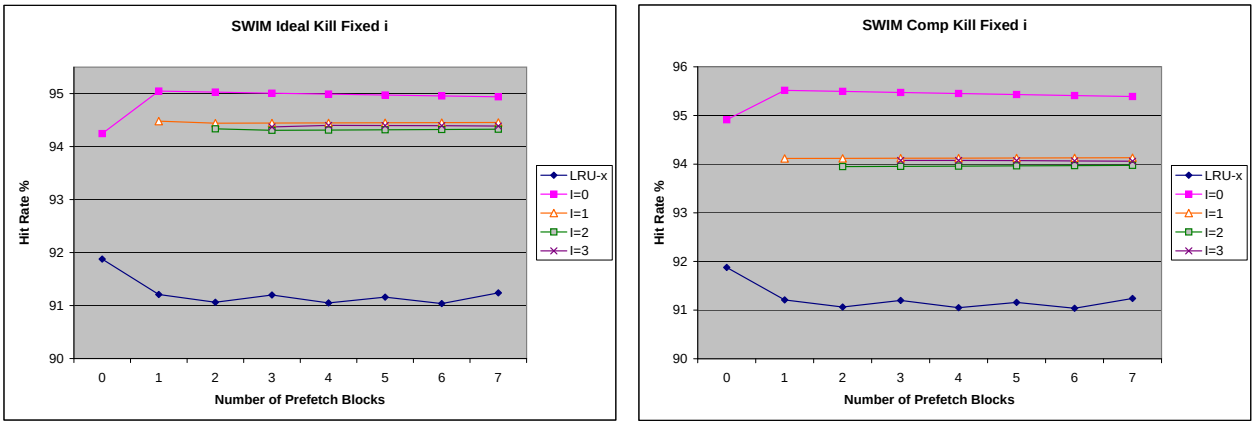


Figure 7: SWIM L1 Hit Rate

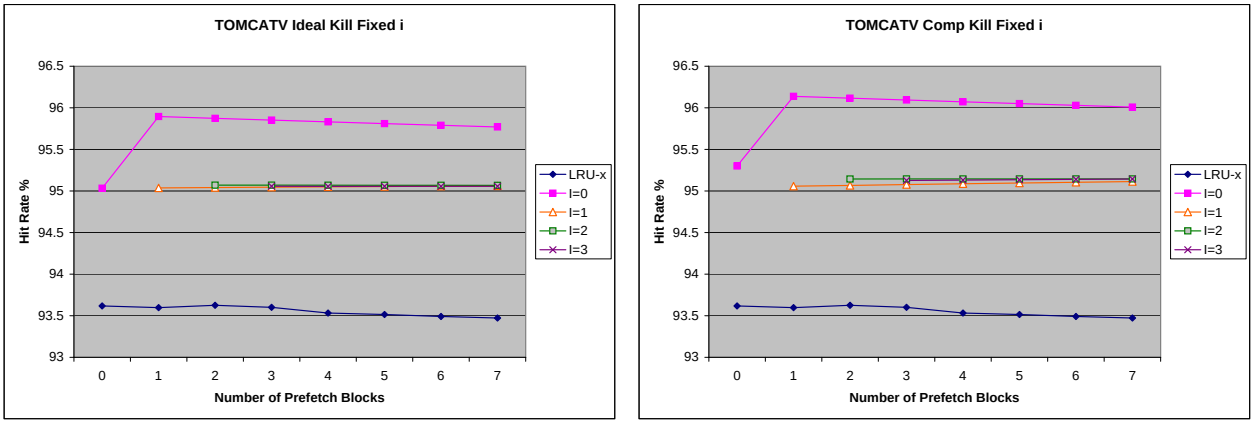
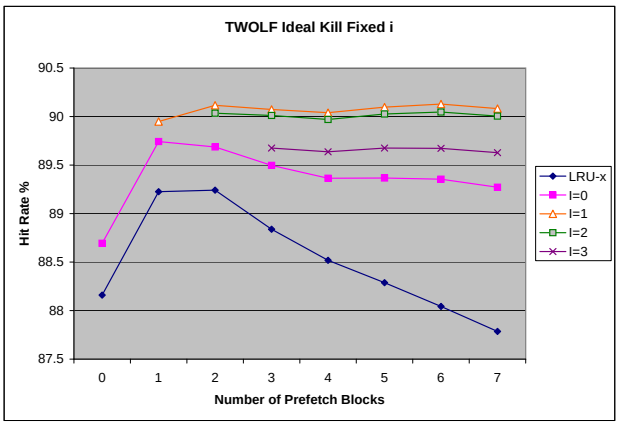


Figure 8: TOMCATV L1 Hit Rate



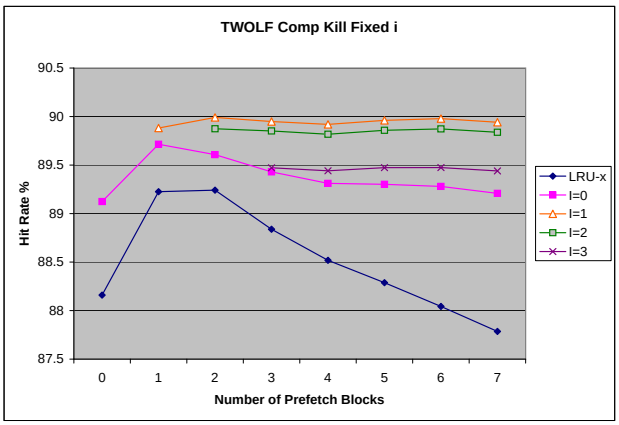
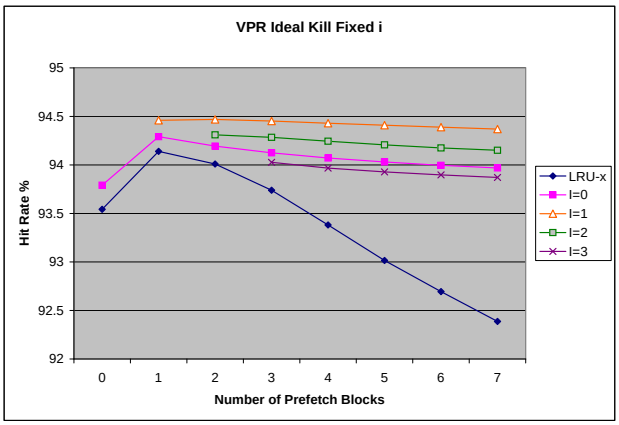


Figure 9: TWOLF L1 Hit Rate



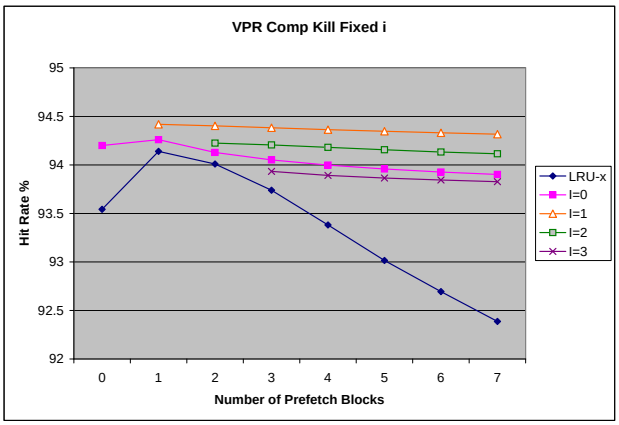


Figure 10: VPR L1 Hit Rate

Benchmark	Ideal Kill Hints	Compiler Kill Hints		% of Ideal Kill Hints
		Load-Store	Range	
ART	39943194	20021095	67516	50.29
BZIP	4286636	2562281	195183	64.32
GCC	5385506	1256474	175201	26.58
MCF	49251433	27548733	306489	56.56
SWIM	7657011	5452208	7120	71.30
TOMCATV	9177531	4685160	17544	51.24
TWOLF	15805947	11063292	82631	70.51
VPR	8207570	5208584	37332	63.91

Figure 11: Kill Hints for Ideal and Compiler-Kill

Benchmark	LRU-0	LRU-1	LRU-2	Ideal (1,1)	Comp (1,1)
ART	75.85	80.51	86.97	88.30	86.89
BZIP	96.28	97.02	96.97	97.20	97.14
GCC	96.49	98.82	98.81	99.09	98.91
MCF	67.38	71.25	67.66	71.75	71.59
SWIM	91.88	91.21	91.06	94.44	94.12
TOMCATV	93.62	93.60	93.63	95.04	95.07
TWOLF	88.16	89.23	89.24	90.33	89.99
VPR	93.54	94.14	94.01	94.50	94.40

Figure 12: LRU-0, LRU-1, LRU-2, Ideal (1,1), and Comp (1,1) L1 Hit Rates

Benchmark	LRU-1	LRU-2	Ideal (1,1)	Comp (1,1)	%MemoryOp
ART	18.37	58.77	70.83	58.11	33.08
BZIP	8.36	7.81	10.64	9.83	23.02
GCC	33.00	32.88	38.25	34.75	30.66
MCF	11.18	0.74	12.80	12.29	30.19
SWIM	-4.55	-5.50	22.42	19.05	18.85
TOMCATV	-0.16	0.06	13.13	13.40	28.65
TWOLF	6.40	6.50	13.95	11.52	26.69
VPR	5.09	3.94	8.39	7.49	25.42
Average	9.71	13.15	23.80	20.80	27.07

Figure 13: LRU-1, LRU-2, Ideal (1,1), and Comp (1,1) Memory Performance Improvement with respect to LRU-0 (no prefetching). L2 latency = 18 cycles.

7 Conclusions and Ongoing Work

Prefetching has been a subject of intensive research. Pure software, pure hardware and integrated software and hardware methods have been proposed. The main contribution of our work is an integration of cache replacement algorithms with prefetching in a generic manner. The cache replacement can be software-assisted as described herein, or can be done in hardware. Prefetching can be done in hardware as described herein, or done in software.

We have shown using analysis and experiments with a parameterizable hardware prefetch method that cache pollution, a significant problem in aggressive hardware prefetch methods, can be controlled by modifying cache replacement. The required bandwidth to cache can be limited. The resultant prefetch strategies improve performance in many cases. Further, we have shown that a new prefetch scheme where 1 or 2 adjacent blocks are prefetched depending on whether there is dead data in the cache or not works very well, and is significantly more stable than standard sequential hardware prefetching.

Ongoing work includes experimentation with more sophisticated prefetching schemes including variable-stride prefetching, and prefetch methods which target reducing the memory performance overhead associated with context switches in timeshared systems.

References

[1] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[2] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, December 1995.

[3] J. P. Casmira and D. R. Kaeli. Modeling Cache Pollution. In *Proceedings of the* 2^{nd} *IASTED Conference on Modeling and Simulation*, pages 123–126, 1995.

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

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

[6] E. H. Gornish and A. V. Veidenbaum. An Integrated Hardware/Software Scheme for Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 281–284, St. Charles, IL, 1994.

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

[8] A. C. Klaiber and H. M. Levy. An Architecture for Software-controlled Data Prefetching. *SIGARCH Computer Architecture News*, 19(3):43–53, May 1991.

[9] An-Chow Lai and Babak Falsafi. Selective, Accurate, and Timely Self-invalidation Using Last-touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[10] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (to appear)*, July 2001.

[11] 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.

[12] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software Prefetching in Pointer- and Call-intensive Environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, pages 231–236, November 1995.

[13] C.-K. Luk and T. C. Mowry. Compiler-based Prefetching for Recursive Data Structures. *ACM SIGOPS Operating Systems Review*, 30(5):222–233, 1996.

[14] 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.

[15] 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.

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

[17] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, MA, October 1992.

[18] Toshihiro Ozawa, Yasunori Kimura, and Shin'ichiro Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-purpose Programs. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, November 1995.

[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of* 15^{th} *Symposium on Operating System Principles*, pages 79–95, December 1995.

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

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

[22] 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.

[23] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 2000.

[24] 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.

[25] Z. Zhang and J. Torrellas. Speeding Up Irregular Applications in Shared-Memory Multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA '95)*, June 1995.

A Appendix

A.1 Lemma 2 Proof

Proof: 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 $C(KIL, t + 1) \preceq_1 C(LRU, t + 1)$. ■

A.2 Theorem 1 Proof

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 $C(LRU, t+1) \preceq_0 C(KIL, t+1)$. ■