Cooperative Caching with Keep-Me and Evict-Me

Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang[*]

Dept. of Computer Sciences, University of Texas at Austin,
Email: {jbsartor, vsubram, mckinley}@cs.utexas.edu
[*]Dept. of Computer Science, Michigan Technological University,
Email: zlwang@mtu.edu

Abstract

Cooperative caching seeks to improve memory system performance by using compiler locality hints to assist hardware cache decisions. In this paper, the compiler suggests cache lines to keep or evict in set-associative caches. A compiler analysis predicts data that will be and will not be reused, and annotates the corresponding memory operations with a keep-me or evict-me hint. The architecture maintains these hints on a cache line and only acts on them on a cache miss. Evict-me caching prefers to evict lines marked evict-me. Keep-me caching retains keep-me lines if possible. Otherwise, the default replacement algorithm evicts the least-recently-used (LRU) line in the set. This paper introduces the keep-me hint, the associated compiler analysis, and architectural support. The keep-me architecture includes very modest ISA support, replacement algorithms, and decay mechanisms that avoid retaining keep-me lines indefinitely. Our results are mixed for our implementation of keep-me, but show it has potential. We combine keep-me and evict-me from previous work, but find few additive benefits due to limitations in our compiler algorithm which only applies each independently rather than performing a combined analysis.

1 Introduction

The gap between processor and memory speed continues to grow and memory accesses increasingly are the main processor performance bottleneck. Furthermore, to attain small memory latencies in future technologies designers are maintaining or shrinking or partitioning caches [3, 24]. Modern memory systems use cache hierarchies with small degrees of set associativity and techniques like hardware prefetching to hide memory latencies [18, 33]. These trends and large data sizes are exacerbating capacity and conflict misses. Unfortunately, the typical ISA interface to this parallel and complex memory system remains entrenched in a 30 year old design as a simplistic load/store *eye dropper* with perhaps a software prefetch. To attain improved performance on next generation memory systems, the ISA will need to broaden its interface. This paper continues in the vein of prior work [12, 43, 44, 46] that suggests using compiler analysis together with modest hardware support to improve memory efficiency and effectiveness. Here, we consider improving set-associative cache decisions.

Set-associative caches typically use an Least-Recently-Used (LRU) replacement policy. When the cache brings in a new line, it evicts the LRU data from the set. For many programs, this policy does not perform well [3, 10, 31]. The current hardware-only approach is inherently limited because it can only ever use the past to predict the future. A software-only approach is also inherently limited because although the compiler can accurately predict data reuse within a procedure, it quickly loses accuracy beyond this scope.

Our research focuses on a cooperative software/hardware approach that strives to combine the best of both static and dynamic information. The compiler provides hints as to which data to keep or evict in set-associative caches. When the hardware is performing well, and program references hit in the cache, it ignores the hints. However on a miss, it uses the hints to guide its replacement decisions. Thus, the hardware defaults to its statistical decisions that use past history and wide scope, but uses compiler guidance when needed and available.

Wang et al. first proposed *evict-me* caching [44]. The evict-me compiler analysis finds array references

This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

in loops that the program will not use again soon, and sets an *evict-me* hint bit on the memory instruction (load/store). The hardware stores the hints on the cache line. On a miss, the hardware preferentially evicts evict-me cache lines rather than the LRU line. If the compiler guarantees the reuse distances of evict-me data are longer than other data, Wang et al. prove this cache will perform better than LRU or at least as well.

This work introduces a complementary cooperative policy called *keep-me*. The keep-me compiler analysis finds array accesses with temporal and spatial reuse that the cache has sufficient capacity to retain, and marks the first memory instruction with keep-me. We experiment with several compiler keep-me heuristics. The hardware acts on this hint on a miss. The keep-me replacement algorithm evicts cache lines *not* marked with keep-me in LRU order. If all lines are keep-me, it defaults to LRU. However, if the cache never evicts keep-me lines, they would eventually monopolize the cache and all replacement would revert to LRU. To gradually decay keep-me, the hardware uses a keep-me counter which it decrements on each replacement. Because the hardware support adds only a few bits to the LRU comparison and performs the comparison only on a miss, it will not change hit or miss times.

We implement our compiler algorithm in the Scale compiler [29] and the hardware component in SimpleScalar [4], a hardware simulator. We use five benchmark programs from SPEC 2000, NAS, and Perfect, and four kernels with selection of modern cache configurations. Our results demonstrate that the effectiveness of cooperative caching is highly dependent on the benchmark, data sizes, and cache configuration. Our best compiler heuristic heuristics for keep-me yields mixed results. In the worst result, keep-me degrades performance by 7%. In the cache configuration with the best average results, keep-me improves performance by 6% for one program, and deteriorates performance by 1% for another. Keep-me and evict-me together improve total simulated performance by 6% to -1%. For another cache configuration, they improve simulated performance by as much as 35.7%. Thus, we find potential for keep-me but have not yet delivered a system that only maintains or improves performance.

Keep-me and evict-me should be more than additive since they reduce misses in different ways. Evict-me improves cache performance by improving replacement decisions due to conflict or capacity misses, whereas keep-me targets capacity misses by explicitly choosing the data to keep in the cache. This work simply combines the algorithms. A better compiler algorithm that explicitly reasons about both at the same time should be able to perform better than the results we show here. We leave that exploration for future work.

```
program toybench
   integer a(2000), b(400,2000),c(400,2000)
   integer p(400, 2000),r(400, 200)
      do j = 1,400
         do k = 1,2000
            p(j,k) = p(j,k) + b(j,k)*a(k)
         enddo

         do m = 1,2000
            r(j,m) = r(j,m) + c(j,m)*a(m)
         enddo
      enddo
end
```

Figure 1. Fortran Example to Illustrate Keep-me

The remainder of this paper is organized as follows. Section 2 presents an example to illustrate the potential of keep-me. Section 3.1 describes the software implementation and a range of heuristics that mark temporal and spatial data. Section 3.2 discusses the hardware modifications to effectively use the compiler information. Section 4 discusses the experimental framework and simulation results. Section 5 discusses related work and then we conclude.

2 Motivating Example

This section presents an example that shows potential miss rate reductions by using keep-me to target temporal loads. In Figure 1, array *a* has temporal reuse between the inner *k* and *m* loops. If there is insufficient capacity or poor choice of replacement, all accesses to *a* can miss. Assuming *a* is less than the cache size, the potential benefit of keep-me is to assure that the *m* loop's accesses to *a* are hits.

Assume a fully associative 8k (8192 bytes) L1 cache with LRU replacement and 4 byte integers. Consider one iteration of the outer (*j*) loop. Since *j* is a constant in the *k* loop, each of arrays *b*, *a*, and *p* touches 8000 bytes, 24000 bytes total which is roughly three times the cache capacity. Initially, the *k* loop will experience a cold misses. When loop volume reaches 8192 bytes, subsequent loop accesses will then replace LRU data (around 16000 bytes), evicting all three arrays roughly equally. In the *m* loop with LRU, there will be capacity misses to *a* and cold misses to *r* and *c*. Thus LRU attains a miss rate of 100%. If the compiler marks the accesses to *a* in the *k* loop with keep-me, the miss rate can drop to 86% (40000/48000).

Keep-me improvements are very dependent on the cache and data size match. If the *k* loop volume is much smaller than the cache, the cache will usually retain *a* without keep-me. If the volume of *a* is much larger than

```
setKeepMeTag() {
  for each loop nest {
    compute NV = nest volume
    for each array reference r in nest {
      if (r has temporal reuse in this nest) {
        if (NV > cache size && NV < 2 * cache size) {
          mark r with keep-me
          set keep-me counter to max value (implicit)
        }
        else if (NV unknown && r has temporal reuse with next loop) {
          mark r with keep-me
          set keep-me counter to max value (implicit)
} } } } }
```

Figure 2. Temporal Keep-me Heuristic Pseudocode

the cache, caching all of *a* is not possible and marking the *k* loop accesses to *a* with keep-me is insufficient. We leave this case for future work.

3 Keep-Me

Keep-me caching requires compiler analysis and architectural support. The compiler uses dependence analysis to detect reuse, and replaces the first load/store to that data with a special keep-me load/store. The hardware support adds keep-me bits to the LRU bits. On a keep-me load/store, it sets the keep-me flag bit and counter bits. On a miss, it uses the bits to select a replacement line. To prevent the cache from keeping this data forever, the replacement algorithm decrements the keep-me counter bits on misses to the set. When the counter reaches zero, the cache reverts to the LRU replacement policy for this line. The remaining sections describes our system, policies, and potential variations in more detail.

3.1 Keep-me Compiler Analysis

Our compiler analysis identifies and marks array references with temporal and spatial reuse with keep-me that it predicts the cache has sufficient capacity to exploit. Table 1 enumerates the compiler heuristics that vary in both the percent of memory instructions they mark, and the reuse distance they attempt to tolerate. It also indicates whether each heuristic targets temporal and/or spatial reuse. We discuss them in logical order, each improving upon the last. All of our results use the *cspatial* heuristic described at the end of the section.

3.1.1 Heuristics

The *temporal* keep-me heuristic identifies data with temporal reuse within the same loop nest. Figure 2 shows the pseudo-code which first computes the loop nest volume of each nest. If an array reference has temporal reuse within a loop nest, the algorithm checks the nest volume. If the nest volume is larger than the cache size, but less than twice the cache size, the compiler marks the array reference with keep-me. If the total nest volume is less than the cache size, the cache will naturally keep all data from the nest. If the nest volume is larger than twice the cache size, there is a chance that reuse distances will be too large to effectively keep the corresponding array volume in the cache.

Often the compiler cannot statically determine loop nest volume because of unknown loop bounds. In this case, the *temporal* algorithm marks the access keep-me if the array reference has temporal reuse with the adjacent loop at the same depth. This heuristic assumes that the volume of the two loops does not greatly exceed the cache size. When this assumption is wrong, the cache will perform poorly regardless, and keep-me should not exacerbate it.

The *temporal* heuristic marks relatively few loads as keep-me. We also tried an aggressive heuristic called *indiscriminate* that marks all loads as keep me if they have any temporal reuse within the nest. This heuristic marks significantly more keep-me memory references compared with the *temporal* heuristic. *Indiscriminate* decreases memory miss rates more than *temporal* for some programs. Unfortunately, it sometimes substantially increases miss rates because it floods the cache with keep-me bits that mask their LRU position.

Trying to find a sweet-spot between the two, we implemented the *cap* heuristic by modifying *indiscriminate*. *Cap* marks array references with keep-me until it reaches a threshold percentage of the loads in the nest. We experiment with thresholds of 25%, 50%, and 75%. *Cap* slightly improves performance over *indiscriminate*, and limits its exposure to large performance degradations. Instead of selecting references on a first-come-first-serve (FCFS) basis as *cap* does, we implemented an algorithm that prioritized references to arrays that it marked keep-me from prior loops. This scheme preferentially marks the arrays with a keep-me history in subsequent loops. If these references are insufficient to satisfy the cap, the compiler reverts to FCFS until it meets the cap. This priority scheme made little difference to the results of *cap*, and thus we do not explore it further.

None of the above heuristics explicitly target data with short distance spatial locality. To determine if keep-me was disturbing this spatial locality which a conventional cache would capture, we introduced the *ispatial* heuristic. *Ispatial* adds short spatial reuse to *indiscriminate*. The compiler thus marks spatial data as keep-me and spatial. The compiler communicates both the keep-me and the spatial keep-me hints to the architecture in memory instructions. The architecture turns off the spatial bit when the program touches the last element in the

Name	(s)patial/ (t)emporal bits	Description
temporal	t	Keep-me is marked on temporal loads based on nest volume
indiscriminate	t	Keep-me is marked on all temporal loads irrespective of nest volume
cap	t	Keep-me is marked on temporal loads up to a maximum of a fixed percentage of loads per nest
ispatial	s/t	Similar to *indiscriminate*. The compiler also marks all spatial loads with spatial keep-me
cspatial	s/t	Combines *ispatial* and *cap*

Table 1. Compiler Heuristics

```
setKeepMeTag() {
  for each loop nest {
    for each array reference r in nest {
      if (r has temporal reuse in this nest) {
        if (less than CAP% of refs in this loop are marked with keep-me) {
          mark r with keep-me
        }
      }
      else if (r has spatial reuse in this nest) {
        if (less than CAP% of refs in this loop are marked with keep-me) {
          mark r with spatial keep-me
} } } } }
```

Figure 3. Cspatial Keep-me Heuristic Pseudocode

line. It thus reduces the time it keeps spatial data as compared with temporal keep-me data. This heuristic eliminated some keep-me degradations when temporal keep-me data interfered with and spoiled spatial reuse.

Our best heuristic combines *cap* and *ispatial* which limits the number of memory instructions set with temporal and spatial keep-me. Figure 3 presents the pseudocode for the *cspatial* heuristic. It combines the improved performance of *cap* over *indiscriminate*, but also insures that keep-me does not disturb short distance spatial locality. The remainder of this paper uses this *cspatial* heuristic with loop cap thresholds of 25%, 50%, and 75%.

3.1.2 Potential Improvements

All our heuristics use the same keep-me decay counter value, as shown in both branches of the inner if-then-else in Figure 2. In the current implementation (see Section 3.2), the hardware sets the counter value based on the set-associativity of the cache. Since the hardware decays the counter on each replacement until it invalidates keep-me, higher counter value will retain the line longer in the cache. Experiments varying this counter value did not change the results much.

However, by adjusting the counter on a per-reference basis, the compiler could differentiate data with short reuse over data with a longer reuse distance. For example, if the nest volume falls between zero and the cache size, we could mark the array reference with keep-me and reduce the initial value of the keep-me counter. These changes would modify the second *if* in Figure 2 and require dedicating instruction bits to pass the maximum counter. We believe this feature may provide a benefit in future investigations.

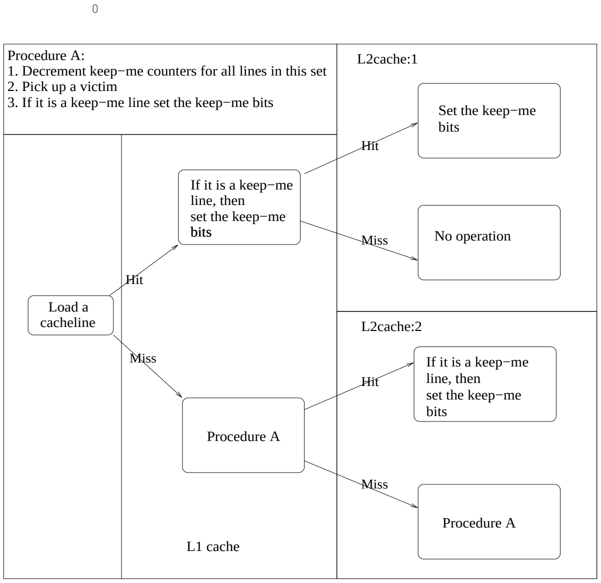

0
Procedure A:
1. Decrement keep−me counters for all lines in this set
2. Pick up a victim
3. If it is a keep−me line set the keep−me bits
L2cache:1
Set the keep−me bits
Hit
If it is a keep−me line, then set the keep−me bits
Miss
No operation
Hit
Load a cacheline
L2cache:2
Miss
If it is a keep−me line, then set the keep−me bits
Hit
Procedure A
Miss
Procedure A
L1 cache

Figure 4. State diagram of L1 and L2 cache behavior in a keep-me cache

3.2 Keep-me Hardware Support

This section discusses keep-me instruction-set architecture (ISA) support, policies for decaying the keep-me counter, and policies for using keep-me in multiple cache levels (we use two levels).

The simplistic load/store memory/system interface is increasingly showing its age. Since these instructions determine more of processor performance, next generation ISAs will need to support specialized versions. This trend is already apparent with the addition of prefetch and write-no-fetch instructions in many architectures. We recommend special keep-me and evict-me instructions with additional bits for spatial, temporal, and counter tags, following Wang et al. [44].

The Alpha ISA [23] and SimpleScalar [4] infrastructure limit our prototype keep-me ISA implementation. We steal three bits from the load offset. (In our benchmarks, fewer than 1% of loads use these bits [42], and we inhibit the hints on them.) We use two bits for keep-me and one for evict-me. Wang et al.'s evict-me implementation used 4 bits: the evict-me flag, a spatial direction bit, and two bits for the spatial stride. A non-zero spatial stride flagged spatial locality. Wang et al.'s evict-me implementation had this extra information and thus performs better with spatial data than this evict-me implementation which must share its bits with keep-me. The previous evict-me evaluation differs also because it used a different ISA and simulator, URSIM [12].

For keep-me, the first bit turns keep-me on, and the second flags stride-one forward spatial reuse. With a spatial keep-me, the hardware tracks cache line hits. When the program accesses the last element in the line, the hardware turns off the keep-me bit. This mechanism insures the cache exploits short distance spatial locality but does not keep the line longer than necessary. If the compiler detects both temporal and spatial keep-me reuse, it does not set the spatial bit.

To decay the keep-me hints, the hardware adds a keep-me counter to each keep-me cache line. Without a counter, a keep-me line could potentially stay forever in the cache or if all lines are keep-me, the replacement algorithm will revert to LRU. If too many lines are keep-me, degradations compared with LRU may result for two reasons. First, the keep-me lines may cause the eviction of other lines, reducing their cache residency time and disrupting their reuse. Second, this increase in misses will decay the keep-me counter more rapidly, and may lead to keep-me lines being evicted before its intended reuse. The hardware initializes a keep-me counter to a value equal to the set-associativity of the cache, but other hardware defaults or software controlled values are possible.

Our keep-me implementation is described in the form a state diagram in Figure 4. The state diagram depicts what happens as a load is serviced through the L1 and L2 caches.

On a hit, the hardware updates the line's LRU position, but takes no other action. On a miss, the hardware decrements the keep-me counters in the set. When the counter reaches zero, the hardware ignores the keep-me bit. At a minimum, this algorithm will keep this line in the cache one replacement longer than LRU. On a miss, the hardware selects a line without keep-me if possible. It first examines the LRU position. If keep-me is set and its counter is non-zero, it inspects the next line in the LRU list. This process continues until it finds a line without keep-me, or defaults to LRU. With evict-me or with both keep-me and evict-me, the replacement algorithm selects the LRU evict-me line if one exists. Since keep-me and evict-me take no action on a hit, they do not increase hit cycle times. On a miss, they require minimal additional logic and can examine replacement bits in parallel, all of which can be accommodated within the miss latency.

Once the counter reaches zero, our hardware implementation treats the keep-me and spatial keep-me lines the same as any other line. Our implementation does not turn off the keep-me bits, because it migrates the information to the next cache level (from the L1 to L2 in our experiments). We implement a mostly inclusive cache that follows the P4's policies [18] in which the hardware places a line in both caches on its first access, but makes independent replacement decisions.* Our implementation sets keep-me in both caches and uses the above policies. However, when the hardware evicts a keep-me line from the L1 that is resident in the L2, it reinitializes the L2 keep-me counter to attain longer distance reuse, if possible. Simulation results show that this migration feature improves keep-me performance. If the line is not resident in the L2, the hardware does not insert it.

We also implement an enhancement on a hit. On a keep-me memory instruction hit, if the line is not set to keep-me, the hardware sets the keep-me bits. We also replicate the hint in the corresponding line in the L2. This logic can proceed independently of the hit service. This enhancement on a hit is described in Figure 4 in the box named L2cache:1.

In addition to the software heuristics that limit the number of keep-me lines, we explored hardware solutions when too many lines are keep-me. We experiment with a bound on the number of keep-me lines in a set. For example, with a 4-way set associative cache and a 50% bound, if two keep-me lines with non-zero counters are already in the set the hardware turns off keep-me in subsequent keep-me lines by setting their counters to zero. We use this implementation instead of simply turning off the keep-me bit, so the hardware can migrate this keep-me line to the next level cache, provided the line is resident and the set has not yet reached its bound.

4 Experimental Results

This section first overviews our compiler, simulator, and benchmarks. It then presents results. We include statis-

*Another possible implementation is the AMD Athlon on-die L2, non-inclusive cache hierarchy in which the hardware puts first accesses only in the L1, but then puts all evicted L1 lines in the L2.

tics on the static and dynamic number of keep-me hints, and show that keep-me sometimes improves and degrades cache hit rates with the expected impact on simulated performance. In one case, combining evict-me and keep-me attains substantial benefits, but on average provides no synergistic benefits in our current implementation.

4.1 Experimental Framework

Compiler Infrastructure. We use Scale, a research compiler infrastructure for Fortran and C program written in Java [29]. Scale uses static-single-assignment and performs many classic compiler optimizations (constant propagation, value numbering, register allocation, alias analysis, etc.). Although it is a research compiler, it achieves competitive performance for the Alpha and Sparc architectures. For example, on SPEC2000 INT on the Alpha, its average performs is a few percent worse than gcc and 35% worse than the native Alpha compiler. On SPEC2000 FP, it achieves performance on average 2.5 times better than gcc, but 2 times worse than the native Alpha compiler. It thus provides a good base experimental platform.

Scale includes an implementation of evict-me. We add the keep-me compiler analysis, as described in Section 3.1. Both keep-me and evict-me require the compiler to generate a data dependence graph [15, 36] and perform regular section analysis [17]. Our dependence testing is based on the Omega library [36, 41]. Keep-me and evict-me introduce hints by embedding them separately in the assembly code output for the Alpha. We post-process the commented assembly code and encode the hints directly into the assembly using unimplemented alpha instructions. We then statically link the assembly code on a native alpha machine and run the code in our simulator.

Simulator Infrastructure. We modified SimpleScalar for this work. SimpleScalar simulates an out-of-order issue processor, with non-blocking caches, speculative execution and branch prediction [4]. SimpleScalar splits the L1 cache into instruction and data caches, and has a shared L2 cache. We model an alpha-like machine with a 1.6 GHZ, 4-way issue, 64-entry RUU (reorder buffer), out-of-order core combined with an effective 800-Mhz, 4-channel Rambus memory system. We model two levels of non-blocking caches, each with 8 miss status handler registers (MSHR). We modify the simulator to recognize keep-me and evict-me hints and handle cache replacement accordingly, including keep-me bits, evict-me bits, and the keep-me and evict-me replacement policies. Simulator flags turn on/off keep-me and evict-me independently in either cache level. Table 2 lists our four cache configurations. We use a modest L2 sizes to increase memory pressure. All of the configurations have the same cache line sizes and latencies. The average latency for memory access is around 200 cycles.

	Con. 1	Con. 2	Con. 3	Con. 4
Level 1	8K, 2-way	16K, 4-way	32K, 2-way	32K, 4-way
	32 byte cache line; 3 cycle latency			
Level 2	128K, 2-way	128K, 4-way	128K, 4-way	256K, 2-way
	128 byte cache line; 12 cycle latency			

Table 2. Cache Configurations

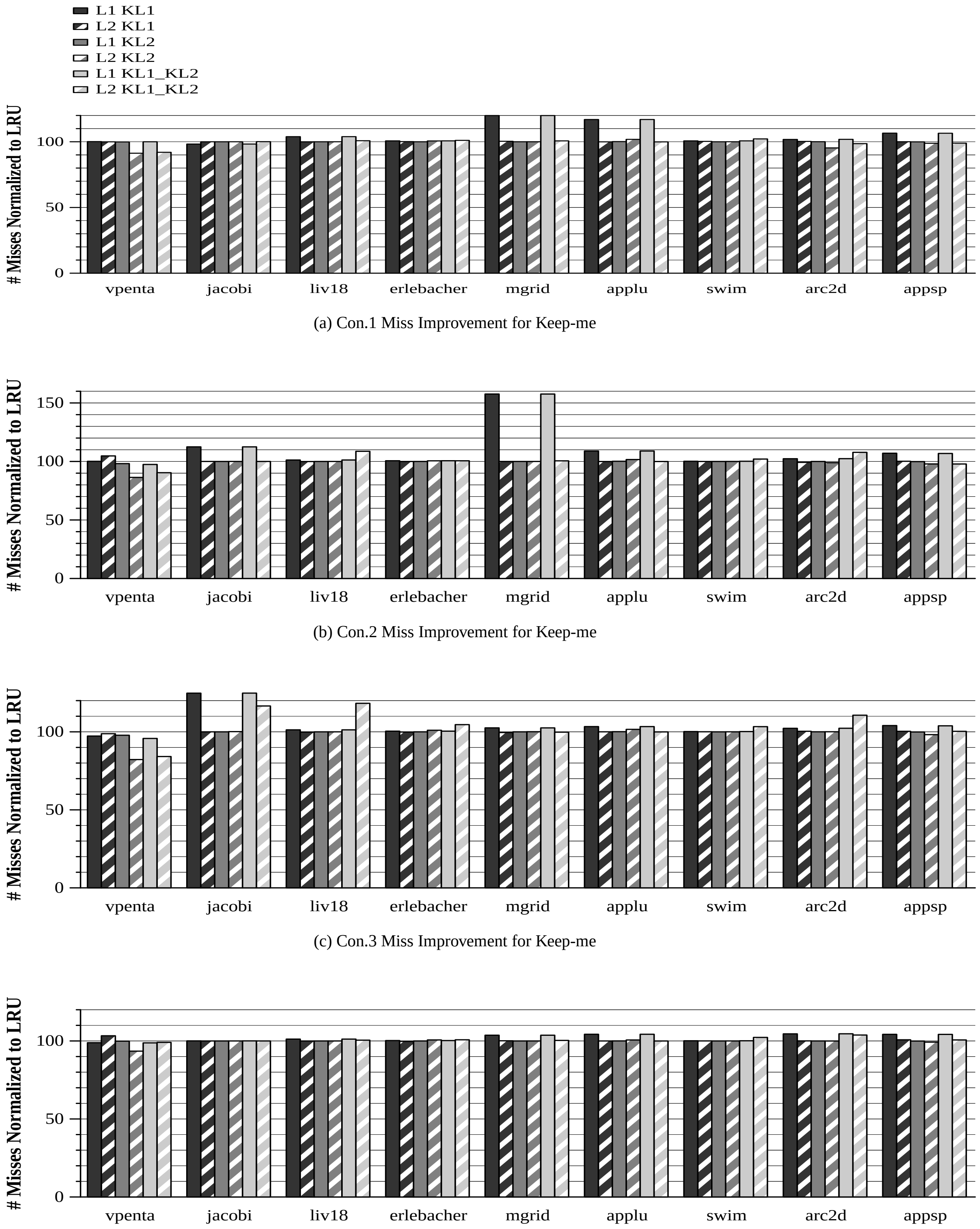
Benchmarks. We use nine benchmarks. Liv18, Vpenta, Erlebacher, and Jacobi are kernels. Swim, Mgrid and Applu are from Spec00. Arc2d is a Perfect benchmark and Appsp is from the Nas Benchmarks. We select benchmarks that lose substantial performance due to data memory stalls (see Figure 6(b)), and benchmarks that Scale compiles. We do not claim these benchmarks are representative.

4.2 Static and Dynamic Counts

This section presents static and dynamic counts of keep-me, and analyzes how keep-me changes cache replacement decisions. Unless otherwise noted, we use a software cap of 75%. We break down keep-me into its temporal and spatial components which indicate how much reuse is available in each benchmark as well as the ratios between temporal and spatial reuse.

Table 3 shows the static and dynamic counts of keep-me hints in Con. 1 (see Table 2). Columns 2, 3, and 4 show the percentage of static memory instructions that Scale marks with keep-me. Columns 5 through 7 and 9 through 11 show the number of memory instructions Scale marks with keep-me as a percentage of dynamic cache accesses in the L1 and L2 respectively. Columns 2, 5, and 9 show the percentage with spatial keep-me, whereas columns 3, 6, and 10 show the percentage with temporal keep-me. Columns 4, 7, and 11 present keep-me hints as a percentage of all memory instructions. Columns 8 and 12 show the percentage of cache replacement decisions that were different from LRU decisions due to keep-me hints.

For almost all benchmarks, the compiler marks a high percentage of static loads and stores with spatial reuse and temporal reuse. Keep-me analysis marks on average 56% of the static memory instructions in loops, reflecting the 75% cap. At runtime, the table shows a wide range in executed keep-me memory instructions. It should be noted that only a very small percentage of memory instructions (2.8%) at runtime have temporal keep-me set when they reach the L2.



(a) Con.1 Miss Improvement for Keep-me

(b) Con.2 Miss Improvement for Keep-me

(c) Con.3 Miss Improvement for Keep-me

(d) Con.4 Miss Improvement for Keep-me

Figure 5. Comparing Keep-me Results with Four Memory Configurations

	Static			Dynamic (Con. 1)							
	spatial keep–me	temporal keep–me	total keep–me	L1 spatial keep–me	L1 temporal keep–me	L1 total keep–me	L1 Repl.	L2 spatial keep–me	L2 temporal keep–me	L2 total keep–me	L2 Repl.
vpenta	7.76	55.33	63.1	27.39	44.24	71.63	21.71	2.38	18.97	21.35	16.61
jacobi	28.57	28.57	57.14	14.91	14.03	28.94	59.63	14.15	0	14.15	0.23
liv18	36.17	34.04	70.21	22.21	13.83	36.05	27.66	24.41	0	24.41	21.87
erlebacher	27.57	15.42	42.99	7.27	14.15	21.43	25.97	15.27	4.76	20.03	11.76
mgrid	15.62	41.14	56.77	3.07	29.92	32.99	23.96	5.55	0	5.55	19.90
applu	32.57	26.68	59.25	14.61	15.19	29.81	16.02	0.29	0.09	0.39	0.03
swim	19.41	22.35	41.76	33.09	27.73	60.83	28.97	16.86	0.02	16.88	10.76
arc2d	21.8	34.52	56.32	14.70	13.38	28.09	23.02	17.02	1.20	18.23	18.81
appsp	25.25	32.45	57.71	17.52	15.42	32.95	26.09	15.18	0.42	15.60	22.80
Average	23.85	32.27	56.13	17.19	20.87	38.08	28.11	12.34	2.82	15.17	13.64

Table 3. Static and Dynamic Keep-Me Statistics

4.3 Effect of Hardware Configuration, Associativity, and Software Cap on Keep-Me

This section compares the number of misses for various cache configurations. We experiment with various cache configurations because we find keep-me is very sensitive to factors such as cache size and associativity, as is evict-me. We examine the effect of the cache size and associativity on misses. We then show our default 75% cap improves over 25% and 50% caps.

The next section presents simulated cycle times, in addition to number of misses for our best configuration, comparing keep-me, evict-me, and their combination. The last section discusses the effect of hardware bounds on keep-me performance. In general, we see very mixed results for keep-me which sometimes improves performance and sometimes deteriorates it.

Figure 5 shows keep-me performance for each memory configuration (see Table 2). We present the percentage improvement of keep-me over LRU with respect to the absolute number of misses for a software cap of 75%. For each benchmark the graph presents the misses for the L1 cache and L2 cache for three keep-me variants: use keep-me (1) only in the L1, (2) only in the L2, or (3) in both the caches. The same results are present in a tabular form in table 6 in the Appendix. Unfortunately, the *cspatial* keep-me heuristic does not always reduce the number of cache misses.

For *mgrid*, *applu* and *appsp*, restricting keep-me to the L1 cache tends to increase misses for small L1 caches. Better performance is obtained in a larger L1 cache (Con. 3 and Con. 4). In small L1 caches, the keep-me heuristic is not managing the L1 cache well or has little effect on it. We believe a more careful orchestration of keep-me and evict-me, perhaps augmented with runtime volume information, may improve over these results.

Restricting keep-me to the L2 cache has little impact on most programs, however *vpenta* improves between 6% and 18% on all configurations. The large L2 size (Con. 4) is basically insensitive to keep-me. Our analysis focuses on reuse distances that are likely less than the L2 cache size, which these results confirm. Using the keep-me policy in both the L1 and L2 caches does not perform as well as L2 only because the poor L1 results continue to degrade the L1 and further degrade L2 replacements. However, the results are highly dependent on the combination between a benchmark, its data set size, and cache configuration.

Figure 5 shows a few benchmarks experience large miss increases. For instance, *mgrid* experiences an increase in L1 misses of over 50% in Con. 2. We examined mgrid and found that *mgrid* has a large percentage of dynamic spatial loads. When the compiler marks too many spatial loads with keep-me, it renders keep-me on temporal loads ineffective. Although the temporal loads will stay in the cache longer, it is not long enough to attain their reuse. To test this theory, we set spatial keep-me counters to zero and left temporal counters equal to the cache associativity. This change eliminates large performance degradations. For instance, with keep-me set in L1 and L2 for For Con. 2, mgrid degrades 3.5% compared to the prior 57% degradation. We need to develop a keep-me compiler algorithm that can make this decision systematically.

The memory configuration that performs the best is the 2-way set-associative L1 and L2 (Con. 1). This result seems to be counter-intuitive as we would expect keep-me to perform better with a larger set-associativity since it provides more replacement choices. Table 3 shows however that keep-me influences only 28% of dynamic replacement decisions in a two-way set-associative L1 cache, and thus on average, each set typically contains only one or no keep-me lines. Thus, increasing set associativity will not help to discriminate among multiple keep-me lines and as the results bear out, does not improve keep-me performance with our current compiler heuristic.

All of the results use a cap of 75%, but we also experimented with a cap of 25% and 50%, and found that on average a cap 25% and 50% perform worse because they do not mark keep-me as aggressively.

The performance of our keep-me policies is highly sensitive to benchmark, cache configuration and cap combinations. Although, we demonstrate some potential for keep-me, we need a better keep-me heuristic that suffers no pathologies.

4.4 Keep-Me, Evict-me, and the Combination

This section presents simulated cycle times for the best configurations of keep-me. It also compares keep-me, evict-me, and their combination. We use Con. 1 and set keep-me only in the L2 cache where it achieved its lowest miss rates. We analyze number of misses normalized to LRU. We also examine the cycle count for each of these normalized to LRU, and present the percent of cycle time lost to memory latency, also normalized to LRU. Overall, keep-me and evict-me together improve performance better than keep-me alone. When we combine keep-me and evict-me, the compiler applies each independently. However, we believe the compiler should orchestrate them together to attain better results.

Figure 6(a) compares the L2 misses of keep-me, evict-me, and their combination, all normalized to the LRU replacement policy. Figure 6(b) shows their simulation cycles normalized to LRU. Figure 6(b) also breaks down the total simulation cycles into the cycles spent waiting for L1 and L2 misses and other cycles. The top parts of the bars show that the performance lost due to L1 and L2 memory stalls is substantial, and furthermore our techniques are not eliminating all these stalls.

With the current trend of increasing L2 memory latencies, the potential performance gains obtained from eliminating L2 misses is substantial. For *vpenta*, keep-me decreases L2 misses by 8.7% over LRU and obtains an improvement of over 6% in cycle time. Even if there is a slight decrease in the corresponding L1 performance, L2 performance usually dominates. Since the L1 misses do not vary much, we omit them here.

Figure 6(a) shows that evict-me does not always perform better than LRU, contrary to previous results [44]. For example, *erlebacher* and *liv18* deteriorate by 9% and 5% respectively. This result is due to a limitation in our evict-me implementation. Since we need bits for keep-me and are limited by the Alpha ISA, we gave evict-me only 1 bit whereas it performs best with 4 bits which include spatial reuse and stride information (see Section 3.2). Evict-me uses spatial stride bits to insure the line first satisfies its spatial needs before eviction. Without them, the hardware will occasionally act on an evict-me hint precipitously.

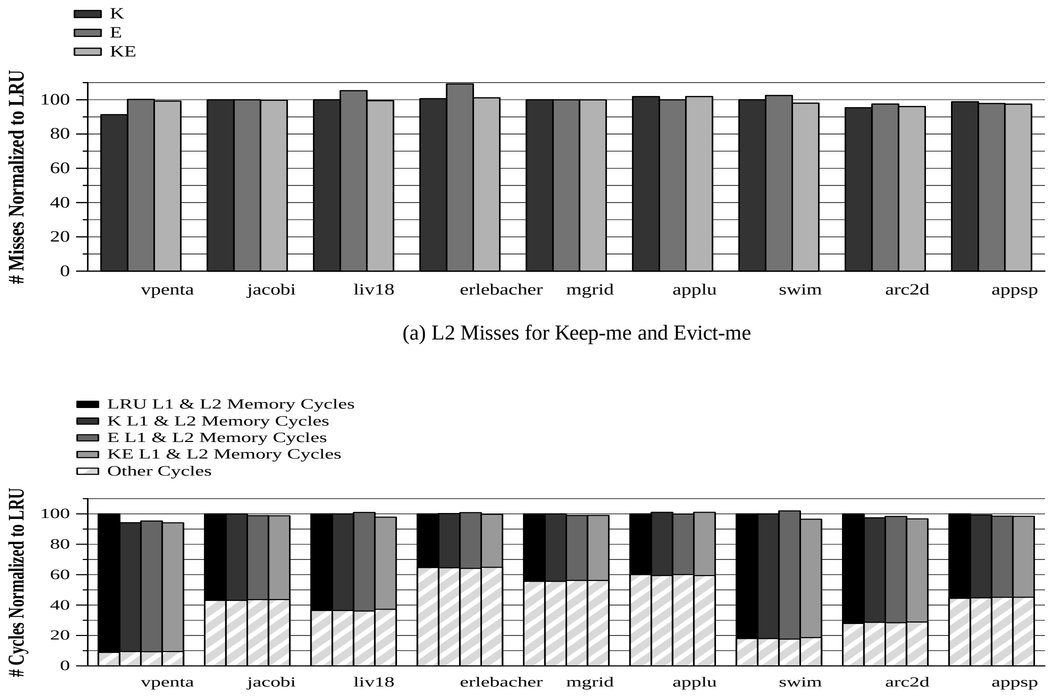
benchmark	cache	KL1	KL2	KL1KL2
vpenta	L1	-0.66	0.99	1.53
vpenta	L2	-5.01	20.73	17.40
jacobi	L1	-0.19	0	-0.19
jacobi	L2	0	0	0
liv18	L1	-1.34	0	-1.34
liv18	L2	0	0	0
erlebacher	L1	1.77	0	1.77
erlebacher	L2	-0.05	0.01	-1.3
mgrid	L1	-25.88	0.00	-25.88
mgrid	L2	-0.14	-0.07	-0.41
applu	L1	-1.12	0	-1.12
applu	L2	0	0	0
swim	L1	0	0	0
swim	L2	0.01	-0.02	-0
arc2d	L1	3.80	0.01	3.81
arc2d	L2	5.05	0.10	2.95
appsp	L1	-0.94	0.09	-0.88
appsp	L2	0.31	2.24	1.35

Table 4. Miss Improvement Percentages for Con. 2, 75% Cap, 50% Hardware Bound with Increased Counters

As we compare the percentage improvement based on regular misses in the L2 cache, we see that keep-me and evict-me together (KE) perform at least as well as the best of keep-me or evict-me in all but *vpenta*. In *vpenta*, however, KE performs slightly better than LRU. For all of the benchmarks except *applu*, KE performs better than evict-me alone. We obtain our best cycle time improvement of 35.7% on *vpenta* using keep-me and evict-me in both the L1 and L2 caches with Con. 3 and a 50% cap. (This configuration is not best on average for all benchmarks though.) This improvement comes from reducing L1 misses by 9.9% and L2 misses by 43.7%. These results demonstrate the potential for keep-me and evict-me together to dramatically improve memory performance.

4.5 Hardware heuristics

This section briefly describes using the hardware bound to control keep-me. The hardware bound prevents a set from degrading to LRU when all the lines are keep-me (see Section 3.2), or degrading further if many of the lines are keep-me. A 25% hardware bound on Con. 2 which is 4-way set associative completely eliminates the degradations shown in Figure 5 and Table 6 improving average performance. We believe this improvement comes from two sources. (1) The bound prevents keep-



(a) L2 Misses for Keep-me and Evict-me

(b) Simulation Cycles for Keep-me and Evict-me

Figure 6. Comparing Keep-me, Evict-me, and the Combination

me from completely obscuring the LRU position. (2) It prevents the hardware from trying and failing to keep too many cache lines by selecting only (in this case) two to keep. Currently we do not distinguish spatial and temporal loads for the bounds, but believe addressing this problem will further improve performance.

To attain reuse beyond one replacement with the keep-me counter, we experimented with increasing its value from 4 to 8 on Con. 2. Increasing the counter either requires additional bit(s), or could be implemented by only decrementing the counter when the line is in the LRU position, instead of on every miss to the set. Table 4 shows the number of misses for a hardware bound of 50% with a temporal keep-me counter of 8 for Con. 2 at a software cap of 75%. This configuration eliminates poor miss behavior, with a few modest improvements. Reducing the hardware bound to 25%, reduces the degradations. For example *mgrid* shows a marginal deterioration of 1.72% for the L1 from about 26% previously. However the maximum gains obtained in this configuration are also reduced. For instance *vpenta* shows a L2 miss rate improvement of 17.4% with a 50% bound, shows only a 8.9 % improvement with the 25% bound. Table 5 in the Appendix shows the results for a hardware bound of 25% for Con. 2. A 50% bound on Con. 2 improves helps this 4-way cache. However, putting a 50% bound on the 2-way Con. 1 reverts to the mixed results obtained without a hardware bound since the software cap generally limits conflicts in this case.

5 Related Work

This section briefly overviews related work on replacement algorithms, cache design, prefetching, and compiler algorithms for improving locality.

5.1 Replacement Algorithms

Early work studied the limits of cache replacement algorithms using program traces. Belady [7] pioneered this area by comparing random cache replacement, LRU, and an optimal algorithm that looks into the future. Sugumar and Abraham [39] used Belady's algorithm to characterize capacity and conflict misses. Temam [40] extended Belady's optimality result by simultaneously exploiting spatial and temporal locality. These studies seek to understand cache characteristics rather than to implement a real cache and related algorithms since the architecture cannot peer into the future.

The Early Eviction LRU (EELRU) [38] paging algorithm motivates and indicates additional potential for keep-me. EELRU improves over LRU when data sizes slightly exceed cache size. It chooses to evict the LRU page or the e^{th} most recently used page. Reference history determines e, the *early eviction point*, a point that is expected to be accessed farther in the future than the LRU position. A similar implementation for caches is too expensive because the high overhead of maintaining the reference history. Our approach offers a lightweight mechanism in which a variant of our compiler analysis could generate early eviction points.

Previous work on evict-me uses static compiler analysis to predict which data the program will not reuse soon and prefers to evict that line on a replacement [44]. Our approach is complementary and builds on evict-me. Keep-me predicts which data will be used in the near future and prefer to keep it in the cache. Evict-me alone will only try to throw out data that has no potential for reuse. It does not directly guarantee that the cache keeps data with reuse. Thus cooperative caching can have two goals: (1) to evict data which has no reuse using evict-me and (2) to retain data the program does reuse with keep-me.

Yang et al. evaluate an evict-me style approach using the 'nt' (non-temporal) hint on the Itaniam architecture. They generally improve matrix multiply, *vpenta*, and *tomcatv* depending on the cache configuration, but occassionally degrade performance [46]. Their compiler algorithm uses bin packing and explicitly models cache occupancy. We instead use reuse to guide our hints. Perhaps their more accurate model of cache occupancy is what cooperative caching needs to achieve consistant improvements.

5.2 Cache Hardware Design

Numerous hardware techniques have been proposed to reduce cache misses [2, 19, 20]. The victim cache was originally designed to enhance direct-mapped caches [20] aimed at reducing conflict misses closely spaced in time. It is probabilistic, rather than predictive. Johnson et al. propose a run time spatial locality detection mechanism [19]. They use a hardware table to keep track of spatial locality dynamically. The fetch size can be varied depending on the spatial locality of fetched data. Their work does not address cache replacement. Rivers et al. use a hardware history to track reuse at run time and to categorize accesses as temporal/non-temporal and cacheable/non-cacheable [37]. Lai et al. use a hardware history table to predict when a cache block is dead and which block to prefetch to replace the dead one [25]. Our technique is based on static compiler analysis and does not require substantially additional hardware.

Researchers have explored many statistical techniques to reduce interference in set associative caches such as reactive-associative, skewed associative, column-associative, and other cache designs [2, 6, 9, 11, 22, 35]. These designs seek to combine the hit *time* of direct-map and the hit *rate* of set associativity. These caches change where data is mapped, instead of explicitly guiding replacement. Evict-me and keep-me are analytical rather than statistical.

McKee et al. propose a stream buffer to bypass stream-like data [28]. We mark stream data as *spatial keep-me*. Our solution works on cache replacement directly and does not require an extra buffer. The Intel IA-64 provides instructions to control caching [13]. The non-temporal load/store bypasses the cache to avoid cache pollution due to streaming data. IA-64 supports locality hints used by prefetch, load, and store instructions to control placements of cache lines in either a "temporal structure" or "non-temporal structure". The hints do not direct cache replacement.

Wong and Baer target reducing temporal misses in the L2 cache [45], but this requires profiling while keep-me analysis is entirely a cooperative effort between the compiler and the hardware and does not require profiling.

Our work takes a different approach than hardware and software data prefetching [5, 20, 27, 32, 34]. Data prefetching tries to fetch data which will be used in the near future to reduce miss penalties. Keep-me tags instead predict which data in the cache the program will use again in the near future, and keep them in the cache. Keep-me does not bring new data into the cache and thus does not have higher bandwidth and other overheads of prefetching. Prefetching can pollute the cache when it brings in useless data unlike evict-me and keep-me. Alpha's *prefetch and evict-next* instruction loads a line into the level 1 cache and evicts it on the next miss to the cache set [23]. Implementing an instruction such as *prefetch and keep-next* will not be very effective because temporal loads typically have larger reuse distances.

Abu-Sufah was the first to have the idea of a software managed cache in order to improve the performance of virtual memory [1]. Recently, more radical cache designs such as Hallnor and Reinhardt propose a software managed cache to reduce DRAM latencies [16]. Region-Based Caching adds to the L1 cache small caches specifically for stack and global data in order to reduce power [26]. Although the cooperation between the compiler and architecture is important to these cache designs, they require more hardware support and focus on different aspects of performance as compared with keep-me. Keep-me focuses on miss rate improvement and cycle time reduction. However keep-me could work in these more radical caches.

5.3 Cache Performance Evaluation

Previous studies found that although conflict misses dominate, capacity misses between loop nests are a significant source of misses [31]. They find most inter-nest misses are temporal in nature. Keep-me targets this temporal reuse.

Bhandarkar and Ding point out that during L2 misses on the Pentium Pro that the CPU can exhaust machine resources causing back pressure on earlier pipeline stages [8]. A substantial portion of program stalls are a consequence of these indirect and the direct effects of L2 cache misses. For this reason, we make a point to pass keep-me hints to the L2 cache.

5.4 Compiler Locality Analysis

Our reuse analysis is based on dependence testing [15, 36] and regular sections [17], but may benefit from analyses such as Ghosh et al.'s miss equations that precisely compute cache misses [14]. Researchers have also used data dependence analysis for loop and data transformations to improve data locality by moving temporal reuse closer together in time and by introducing spatial locality [1, 21, 30]. These algorithms are synergistic with keep-me and should improve its effectiveness further.

6 Conclusion

Technology trends will cause programs to lose more and more performance to memory latencies. Even in current technologies, programs suffer substantial performance penalties due to cache misses. We present a cooperative approach for keep-me caching with a range of compiler and hardware heuristics that vary from highly conservative to highly aggressive. We also explore the interplay of keep-me with evict-me caching. Overall our implementation of keep-me produces mixed results, leaving open to future work the design of cooperative caching policies that suffer no pathologies yet obtain the best demonstrated improvements.

7 Acknowledgements

This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

References

[1] W. A. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[2] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993.

[3] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, Vancouver, Canada, June 2000.

[4] T. Austin and D. Burger. Micro-30 SimpleScalar tutorial. Technical report, http://www.cs.wisc.edu/ mscalar/ss/tutorial.html, 1997.

[5] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, Albuquerque, NM, Nov. 1991.

[6] B. Batson and T. N. Vijaykumar. Reactive associative caches. In *The 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 49–60, Bacelona, Spain, Sept. 2001.

[7] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):79–101, 1966.

[8] D. Bhandarkar and J. Ding. Performance characterization of the pentium pro processor. In *Third International Symposium on High Performance Computer Architecture*, pages 288–297, San Antonio, Texas, Feb. 1997.

[9] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 265–274, June 1995.

[10] D. Burger, A. Kägi, and J. R. Goodman. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 78–89, Philadelphia, PA, May 1996.

[11] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Second International Symposium on High Performance Computer Architecture*, pages 244–253, Feb. 1996.

[12] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Fifth International Symposium on High Performance Computer Architecture*, Orlando, FL, Jan. 1999.

[13] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 31(7):24–32, July 1998.

[14] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[15] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

[16] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, Vancouver, Canada, June 2000.

[17] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[18] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1 2001), 2001.

[19] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 57–64, Research Triangle Park, NC, Dec. 1997.

[20] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[21] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 306–313, Paris, France, Oct. 1998.

[22] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associatlvity. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 131–139, Jerusalem, Israel, June 1989.

[23] R. E. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. Technical report, http://www.compaq.com/AlphaServer/download/ev6chip.pdf, Nov. 1999.

[24] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct. 2002.

[25] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, Goteborg, Swenden, June 2001.

[26] H. H. Lee and G. S. Tyson. Region-based caching: an energy-delay efficient memory architecture for embedded processors. In *Proceedings of PACM (CASES'00)*, San Jose, California, Nov. 2000.

[27] W. Lin, S. K. Reinhardt, and D.Burger. Reducing dram latencies with an integrated memory hierarchy design. In *Seventh International Symposium on High Performance Computer Architecture*, pages 301–312, Monterrey, Mexico, Jan. 2001.

[28] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, and A. P. Batson. Smarter memory: Improving bandwidth for streamed references. *IEEE Computer*, 31(7):54–63, July 1998.

[29] K. S. McKinley, J. Burrill, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weems. The scale compiler. Technical report, 2005. http://ali--www.cs.umass.edu/∼scale/.

[30] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[31] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999.

[32] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, Oct. 1992.

[33] F. P. O'Connell and S. W. White. Power3: The next generation of PowerPC processors. *IBM Journal of Research and Development*, 44(6), 2000.

[34] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, Apr. 1994.

[35] J. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 240–250, Oct. 1998.

[36] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[37] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 449–456, Melbourne, Australia, July 1998.

[38] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 122–133, Atlanta, GA, May 1999.

[39] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 24–35, Santa Clara, CA, May 1993.

[40] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 48(2):150–158, Feb. 1999.

[41] Univerisity of Maryland. *The Omega Library*, 1996. http://www.cs.umd.edu/projects/omega/.

[42] Z. Wang. *Cooperative Hardware/Software Caching for Next-Generation Memory Systems*. PhD thesis, Dept. of Computer Science, University of Massachusetts, Amherst, Dec. 2003.

[43] Z. Wang, D. Burger, S. K. Reinhardt, K. S. McKinley, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, San Diego, California, June 2003.

[44] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *The 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–208, Charlottesville, Virginia, Sept. 2002.

[45] W. A. Wong and J. Baer. Modified LRU policies for improving second-level cache bahavior. In *Sixth International Symposium on High Performance Computer Architecture*, pages 49–60, Toulouse, France, Jan. 2000.

[46] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu. Compiler-assisted cache replacement: Problem formulation and performance evaluation. In *Proceedings of the Sixteenth Workshop on Languages and Compilers for Parallel Computing*, pages 131–139, College Station, Texas, 2003.

8 Appendix

Table 5 shows the Miss percentage improvement for a hardware bound of 25%, software cap of 75% for Con. 2. The temporal counters used for this experiment are equal to twice the cache associativity.

Table 6 shows results for keep-me across the four cache configurations (see Table 2) with a 75% cap. We present the percentage improvement of keep-me over LRU with respect to the absolute number of misses. Columns 3, 6, 9, and 12 show results for turning on keep-me only in the L1 cache. Columns 4, 7, 10, and 13 similarly show results for keep-me in the L2 cache alone. Columns 5, 8, 11, and 14 show results for keep-me in both cache levels. Unfortunately, the *cspatial* keep-me heuristic does not always reduce the number of cache misses.

benchmark	cache	KL1	KL2	KL1KL2
vpenta	L1	-0.34	0.51	1.43
vpenta	L2	-3.05	10.19	8.90
jacobi	L1	-0.19	0	-0.19
jacobi	L2	0	0	0
liv18	L1	-1.73	0	-1.73
liv18	L2	0	0	0
erlebacher	L1	0.89	0	0.89
erlebacher	L2	-0.03	0.00	-0.03
mgrid	L1	-1.72	0.00	-1.72
mgrid	L2	-0.06	-0.03	-0.07
applu	L1	0	0	0
applu	L2	0	-0	0
swim	L1	0	0	0
swim	L2	0.00	-0	-0
arc2d	L1	2.69	0.00	2.69
arc2d	L2	2.18	-0	0.96
appsp	L1	-0.43	0.05	-0.42
appsp	L2	0.20	1.63	1.01

Table 5. Miss Improvement Percentages for Con. 2, 75% Cap, 25% Hardware Bound with Increased Counters

		Con. 1			Con. 2			Con. 3			Con. 4		
benchmark	cache	KL1	KL2	KL1KL2	KL1	KL2	KL1KL2	KL1	KL2	KL1KL2	KL1	KL2	KL1KL2
vpenta	L1	-0.14	0.12	-0.01	-0.12	1.84	2.56	2.70	2.18	4.23	1.13	0.11	1.23
vpenta	L2	0.04	8.71	7.99	-4.74	13.62	9.60	1.20	17.76	15.83	-3.31	6.53	0.91
jacobi	L1	1.73	0	1.72	-12.48	0	-12.48	-24.78	0	-24.81	-0.04	0	-0.04
jacobi	L2	0	0	-0.09	-0	-0	-0	-0	-0.12	-16.56	-0	-0	-0
liv18	L1	-3.88	0	-3.89	-1.23	0	-1.24	-1.28	0	-1.27	-1.19	0	-1.19
liv18	L2	0.01	0	-0.79	0.06	0	-8.67	0.13	0	-18.28	0.01	0	-0.5
erlebacher	L1	-0.68	0.00	-0.69	-0.61	0	-0.61	-0.43	0	-0.43	-0.25	-0	-0.25
erlebacher	L2	-0.12	-0.62	-1.06	0.02	-0.54	-0.57	0.10	-1.01	-4.61	0.23	-0.64	-0.75
mgrid	L1	-19.96	-0	-19.95	-57.63	-0	-57.62	-2.54	-0	-2.53	-3.69	0	-3.68
mgrid	L2	-0.37	-0.01	-0.67	-0	-0	-0.53	0.45	-0.07	0.21	-0.1	-0	-0.34
applu	L1	-16.94	-0.14	-16.94	-8.92	-0.16	-8.92	-3.36	-0.09	-3.36	-4.29	-0.02	-4.29
applu	L2	0.06	-1.84	0.08	0.04	-1.64	0.04	-0.01	-1.58	0.03	0.00	-0.62	0.01
swim	L1	-0.68	0.00	-0.68	-0.17	0	-0.17	-0.17	0	-0.17	-0.11	-0	-0.11
swim	L2	-0.28	-0.01	-2.19	0.01	-0.02	-2.06	-0.03	-0.01	-3.32	-0.02	-0	-2.27
arc2d	L1	-1.78	-0.05	-1.86	-2.33	0.00	-2.39	-2.25	-0.03	-2.29	-4.55	-0	-4.59
arc2d	L2	-0.25	4.65	1.42	0.78	1.02	-7.79	-0.34	-0.04	-10.65	-0.03	0.02	-3.79
appsp	L1	-6.57	0.06	-6.49	-7.04	0.11	-6.83	-4.01	0.08	-3.85	-4.24	0.05	-4.21
appsp	L2	-0.03	1.15	1.04	-0.16	2.13	2.16	-0.33	1.80	-0.31	-0.74	0.75	-0.64

Table 6. Percentage Miss Improvements with Keep-me