Combining Cooperative Software/Hardware Prefetching and Cache Replacment

Zhenlin Wang[†] Kathryn S. McKinley [§] Doug Burger [§]

[†]Dept. of Computer Science
Michigan Technological University
zlwang@cs.mtu.edu

[§]Dept. of Computer Sciences
The University of Texas at Austin
{mckinley, dburger}@cs.utexas.edu

Abstract

Data prefetching is an effective technique to hide memory latency and thus bridge the increasing processor-memory performance gap. Our previous work presents guided region prefetching (GRP), a hardware/software cooperative prefetching technique which cost-effectively tolerates L2 latencies. The compiler hints improve L2 prefetching accuracy and reduce bus bandwidth consumption compared to hardware only prefetching. However, some useless prefetches remain to degrade memory performance. This paper first explores a more aggressive GRP prefetching scheme which pushes L2 prefetches into the L1, similar to the IBM Power 4 and 5 cache designs. This approach yields some additional performance improvements. This work then combines GRP with evict-me, a compiler-assisted cache replacement policy. This combination can reduce cache pollution introduced by useless prefetches and further improve memory system performance.

1 Introduction

Memory system remains a key to the performance of modern architectures. Two previous techniques, *guided region prefetching* (GRP) and *evict-me* caching improve cache performance using a hardware/software cooperative approach [34, 35]. The compiler marks good candidates for prefetching or replacing in set associative caches, and misses to the marked data causes the architecture to act on this information. Thus, if the caches are performing well (i.e., accesses hit), the architecture ignores the hints, but when the caches miss, the architecture uses the compiler guidance to help refine prefetching and cache replacement decisions. This work investigates the combination and interaction of these two techniques, and the utility of pushing data into the L1 cache of prefetched data, similar to the IBM Power 4 and 5 prefetching designs [24, 32].

We begin with SRP, scheduled region prefetching, which is an aggressive L2 region prefetch mechanism [19]. On a miss, SRP generates prefetches of the large region of lines around the miss. It services them only when no demand misses are outstanding, gives priority to the lines near the miss, and sets the replacement bits on the fetched lines to the lowest priority (LRU–least recently used). GRP extends an aggressive hardware-only prefetching by introducing compiler hints to control the prefetching engine [34]. GRP prefetches only on the misses that compiler marks as having spatial reuses. GRP thus reduces prefetch bandwidth needs and improves its accuracy. We observe that a better cache replacement policy can reduce the side effects of useless prefetches and displacement of simultaneously useful data.

Evict-me caching uses the compiler to help control replacement decisions [35]. The compiler marks data as evict-me when it will not be reused in the future, or its reuse distance is larger than the cache capacity. On a miss, the architecture chooses to replace the line whose evict-me bit is set. If no such line is in the replacement set, it defaults to the LRU bits. With the appropriate compiler analysis, this policy always matches or improves over LRU [35].

This paper first investigates using GRP to push prefetch data into the L1 cache. Since the compiler provides accurate GRP prefetches, we can hide additional latency by *pushing* the prefetched lines into the L1 cache. The pushing scheme places additional pressure on cache replacement decisions in both cache levels. The L1 pushes cause write-back replacements into the L2 which place additional pressure on the L2 cache. We investigate a variety of cache replacement policies and show how some alleviate this pressure. In particular, we compare pushing data into LRU or MRU (most recently used) slots, where LRU is the conservative choice, and MRU is effective only when the line is used quickly. In general, using MRU in both the L1 and L2 with GRP and pushing is the best choice. The average improvement is about 2%, but a few programs improved by up to 13%.

We then combine the compiler-guided evict-me cache with GRP and GRP with pushing. The evict-me cache replacement policy helps reduce pollution resulting from prefetching. It reduces cache misses and is thus orthogonal to region prefetching techniques, which tolerate latencies. Since on-demand misses trigger region prefetches, reducing the L2 misses by using evict-me will also reduce total prefetches as well as total memory traffic. However, the combination does not necessarily bring additional performance gains if region prefetching has already hidden these same latencies. We find that the combination reduces cache miss rates, but due to inaccuracies in our L1 cache model our simulation improvements are small. We intend to fix these inaccuracies the near future.

The remainder of this paper is organized as follows. Section **??** briefly overviews how GRP and evict-me caching work. We then describe the L1 push scheme we use. The next sections present the simulation environment and our evaluation of pushing, modifying the replacement bits, and combining GRP and evict-me.

2 Background

In this section, we briefly introduce how GRP and evict-me caching work.

2.1 Guided Region Prefetching

Guide region prefetching is built upon an aggressive hardware prefetching technique, called *scheduled region prefetching* (SRP). Scheduled region prefetching (SRP) aggressively exploits spatial locality by attempting to prefetch large (4 KB) memory regions on each L2 cache miss [19]. The two negative effects of aggressive prefetching—memory bus contention and cache pollution—are addressed directly by reducing the priority of prefetches in memory bus request scheduling and in replacement decisions, respectively. However, manipulating prefetching priority and replacement decisions is not sufficient to negate all the negative effects. GRP alleviates the problem by introducing compiler hints. Rather than prefetching a 4K region on every miss, GRP issues prefetches only when a miss on a load marked as *spatial*. The compiler marks a load as spatial when it has spatial reuse exploitable by the current cache settings. The previous work on GRP develops a series compiler analyses to detect spatial locality and shows that the compiler is able to single out most spatial reuses [34].

2.2 Evict-me Cache

Evict-me caching uses the compiler knowledge to assist cache replacement.Compiler analyses estimate the reuse distance of each reference in a loop and mark the reference as evict-me if its reuse distance is greater than twice of the cache size. Each cache line also contains an additional bit called the evict-me bit. The bit is set when a marked load instruction is executed. On a cache miss, the cache line with its evict-me set is preferred to be replaced. The replacement still follows LRU policy if no lines are set.

The challenge to the evict-me caching scheme lies in the difficulty to calculate reuse distances statically, in particular, for the loop with symbolic bounds. The previous work uses a simple heuristic to estimate the reuse distance based data volume. If a reuse of a reference crosses nests whose total volume is greater than twice of cache size, the reference is marked as evict-me [35]. When the loop bounds are known at compile time, it is easy to calculate the volume. For a loop with symbolic bounds, the other heuristic is applied. If the nested level of a loop is greater than 1, the total data volume of the loop is considered beyond the threshold, i.e., twice the cache size.

3 An L1 Push Scheme

In this section, we discuss our data push scheme, which pushes prefetched L2 cache lines into the L1 cache. We first describe the hardware implementation. We then experiment with various combinations of two different cache placement polices: MRU and LRU.

3.1 Hardware Description

We implement the push engine in the L2 cache controller. It is a pure hardware scheme, but it is implicitly driven by the compiler if the L2 prefetcher is compiler-guided. A *pull-based* prefetcher would follow a request-service-response path. First, the prefetches sends a request to the next lower level of the memory hierarchy. The lower memory will then send the data back when it is ready. A push-based prefetcher is simpler. It needs only a response process: when a prefetched block in the lower level is ready, the prefetcher simply pushes it to the higher level. In our design, the push stream shares a common response queue with the regular responses. The L2 cache line size is usually no smaller than the L1 cache line size. When the L2 cache line is bigger, we break an L2 cache line into units of L1 cache line size and push all the units into the L1 cache.

Following the default LRU cache replacement policy, a pushed or prefetched line will evict the LRU line in a set and be loaded into the MRU slot. By manipulating the LRU bits, we can make the new line reside in the LRU slot, the MRU slot, or other positions in the set. Previous work, which does not use compiler guidance, shows that keeping L2 prefetch lines in the LRU slots yields the best performance [19]. In Section 3.2, we examine the impact of the MRU and LRU placement policies.

One major concern about this push scheme is address translation. In our implementation, the L2 cache is physically indexed and the L1 cache is virtually indexed. We need to translate a physical address to a virtual address when pushing a line into the L1 cache. The translation can be done using an ITLB (inverted translation look aside buffer). In contrast to a TLB, an ITLB is a small cache indexed by physical page addresses and each entry contains a virtual page address. Since the L2 prefetching region size is aligned and no bigger than a page size, all requests of a region sit in a single page. This ensures that all pushes of a region typically yield no more than one ITLB miss.

An alternative technique is to keep track of the virtual addresses of L2 prefetches. We can extract the virtual page address from an L2 demand miss. A region prefetching request is enqueued with this address. When a prefetch is issued, the prefetching MSHR for the prefetch will keep the virtual address and the push engine can use this address when the data is ready for pushing.

3.2 Results of the Push Scheme

For our experiments,We simulate program binaries on a version of sim-outorder [2] with guied region prefetching [34] added to the simulator. We use the Alpha-ISA and configure the simulator as a 1.6 GHZ, 4-way issue, 64-entry RUU (reorder buffer), out-of-order core with 64K 2-way split level one caches and a unified 4-way 1MB level 2 cache. This cache hierarchy is combined with an effective 800-Mhz, 4-channel Rambus memory system. The L1 and L2 latencies are 3 and 12 cycles, respectively. Each cache contains 8 MSHRs. We use the SimPoint [28] tool set to select a representative starting point beyond the program's initialization phase. We simulate for 200M instructions from that point.

We use the same set of benchmarks as in the guide region prefetching paper [34], which involve 16 Spec CPU2000 benchmarks and *sphinx*, a speech recognition application [18].

Push Performance

Table 1 shows our results with the L1 push scheme. For each benchmark, the leftmost five columns list the benchmark name and the IPC for the base case, perfect L1 cache, perfect L2 cache, and GRP using the default LRU placement policy (GRP/LRU). The rightmost five columns are percentage performance improvements over GRP/LRU. On average, the placement policies of the prefetched or pushed lines have a very small impact on performance. The worst case, GRP/LRU plus Push/LRU, is within a half percent of the best case, GRP/LRU plus Push/MRU. Compared to GRP/LRU, GRP/MRU shows little improvement or degrades performance for all but *mcf*, where it improves the performance by 12%. GRP/MRU performs 6.7% worse than GRP/LRU for *art*, although it still improves over the base by 18%. The gap between GRP/MRU and GRP/LRU for *art* comes from a short bandwidth-bounded loop where the prefetches have short reuse distances and placing them into the LRU slots causes less pressure on the L2 cache when the prefetches are useless.

The push scheme brings us an additional 2% performance improvement over GRP/LRU. The best combination, GRP/LRU plus Push/MRU, offers an 11% performance boost for *applu*, 9% for *equake*, and 6% for *art*. For four benchmarks, *wupwise*, *mgrid*, *applu*, and *equake*, the combination of GRP and data pushing is able to beat a perfect L2 cache.

Push Accuracy and Coverage

Push accuracy is the number of used pushed lines divided by the total number of pushes. A pushed line is used if it is hit before its eviction. Since the push scheme is built upon the region prefetcher, we use the miss reduction over GRP as a measurement of coverage. Table 2 lists the L1 and L2 miss rates of GRP/LRU in the leftmost two columns. The miss reduction of GRP/LRU plus Push/MRU is shown in the middle two columns and the rightmost column lists its push accuracy. Because the L1 miss reduction affects the raw L2 miss rates, the L2 miss rates shown in this table are L2 misses over *all* data accesses. The push scheme is able to reduce L1 misses by at least 40% for 7 benchmarks over GRP and up to 87% for *applu*. Although the average miss reduction is 30%, we only see a 2.3% performance improvement for two reasons. First, the L1 gap is smaller: miss reduction at L1 yields less performance gain than at L2. Second, the push scheme causes additional pressure on the L2 cache, in fact increasing the L2 misses by 9% compared to GRP. However, the 9% increase does not cause much performance loss since the L2 miss rates of GRP are typically very low. Push accuracy is 53% on average, lower than the 69% accuracy of GRP on L2. This suggests that there is higher replacement pressure on the smaller L1 cache.

Table 3 shows the coverage and accuracy of the other schemes. The push accuracies are very close across different schemes. The small gap on the average coverage among the four schemes does not reflect that there is large variance on some benchmarks. For example, the miss reduction is 82% for *mgrid* with GRP/LRU plus Push/MRU compared to only 45% with GRP/MRU plus Push/LRU. In general, varied placement policies have more impact on performance and coverage for specific benchmarks than for the overall average.

The push scheme cache does not hide the latencies of those L1 misses that hit the L2 cache. Those misses are mostly L1 capacity misses that are contained by the larger L2 cache. They do not trigger L2 prefetching and thus do not bring in pushed lines, which could hide their latencies. A prefetching engine that triggers prefetches upon L1 misses will solve this problem; we leave this option to future work.

4 Combination of Evict-me and Prefetching

Prefetched cache blocks may pollute the cache if the blocks are useless. Several techniques seek to reduce cache pollution. For instance, hardware can detect stride accesses and selectively prefetch blocks that are expected to be useful [25]. Compilers can help reduce the impact of cache pollution introduced by hardware prefetching in two ways. First, the cache pollution of a prefetched cache block may be harmless if it evicts a block that is marked as evict-me. In this situation, the marked line is probably useless anyway. Second, compilers can use locality analysis to decide when prefetching is necessary. Previous work examined the second option, where the compiler hints are generated to guide an aggressive hardware prefetcher [34]. In this section, we explore the first option. GRP enhances an aggressive hardware prefetcher and tolerates L2 miss latencies. In Section 3, we described a push scheme that hides L1 miss latencies. The evict-me cache replacement policy helps reduce

	IPC				Improvement over GRP/LRU (%)				
	Base	Perf L2	Perf L1	GRP/LRU	GRP/MRU	GRP/LRU Push/LRU	GRP/MRU Push/LRU	GRP/LRU Push/MRU	GRP/MRU Push/MRU
gzip	1.98	2.05	2.09	1.98	0.00	0.00	0.00	0.00	0.00
wupwise	2.29	2.74	2.90	2.71	0.18	3.21	3.17	3.54	3.35
swim	0.70	2.34	3.04	1.51	0.80	-3.64	0.66	-1.92	2.12
mgrid	2.42	2.95	3.08	2.88	0.00	3.19	0.83	3.44	0.90
applu	1.41	2.36	2.67	2.30	0.52	8.34	8.69	11.42	11.82
vpr	1.62	1.97	2.09	1.89	-0.05	2.01	1.37	2.70	1.85
mesa	2.52	2.57	2.61	2.53	0.12	-0.16	0.08	-0.28	0.00
art	0.55	1.44	2.18	0.70	-6.70	5.85	-6.28	6.85	-6.42
mcf	0.15	0.74	2.01	0.22	12.44	0.00	12.90	-0.46	12.90
equake	1.01	1.76	1.96	1.69	0.65	8.24	8.72	9.07	9.43
ammp	1.83	2.11	2.39	1.83	-0.33	0.05	-0.33	0.05	-0.38
parser	1.32	1.80	2.13	1.54	-0.32	3.89	4.15	3.82	4.08
gap	2.78	2.96	2.98	2.87	0.00	0.24	0.24	0.24	0.24
bzip2	1.26	1.59	1.83	1.40	0.07	0.36	0.79	0.57	0.93
twolf	1.23	1.59	2.06	1.23	-0.41	0.08	-0.41	0.00	-0.41
apsi	2.59	2.68	2.70	2.67	0.00	0.23	0.23	0.23	0.23
sphinx	1.36	2.25	2.64	1.45	-1.45	1.66	-0.07	1.72	-0.07
mean	1.33	2.01	2.40	1.62	0.27	1.93	1.95	2.35	2.28

Table 1. Performance impact of the L1 push scheme and placement policies

	GRP/LRU		GRP/LRU + Push/MRU		
	Miss Rate		Coverage		Accuracy
	L1	L2	L1	L2	
gzip	1.03	0.21	0.00	0.00	71.96
wupwise	1.74	0.02	67.82	16.67	52.17
swim	11.23	2.80	48.17	-43.37	95.71
mgrid	0.91	0.07	82.42	-11.43	86.68
applu	3.58	0.10	87.71	-86.73	88.09
vpr	2.24	0.13	35.27	2.29	36.46
mesa	0.35	0.02	-11.43	-4.76	3.45
art	49.59	17.90	11.72	6.69	59.48
mcf	51.87	22.71	-1.08	-0.04	9.53
equake	8.38	0.11	67.78	-14.02	84.49
ammp	4.54	0.46	-0.44	1.09	13.56
parser	5.09	0.50	25.93	-5.58	78.63
gap	0.29	0.08	48.28	-9.33	98.6
bzip2	5.26	0.48	8.56	-6.95	42
twolf	9.03	0.89	-0.55	0.34	5.89
apsi	0.22	0.01	40.91	0.00	65.23
sphinx	3.64	1.43	15.11	1.82	16.13
average	9.35	2.82	30.95	-9.02	53.42

Table 2. Coverage and accuracy of the L1 push scheme, GRP/LRU plus Push/MRU

	GRP/LRU Push/LRU			GRP/MRU Push/LRU			GRP/MRU Push/MRU		
	L1	L2	Accu	L1	L2	Accu	L1	L2	Accu
gzip	0.00	0.00	71.43	0.00	0.00	73.79	0.00	0.00	71.15
wupwise	58.62	11.11	44.10	56.90	22.22	44.48	64.37	22.22	51.79
swim	44.61	-42.90	85.68	45.33	-12.30	91.18	48.62	-12.44	97.58
mgrid	78.02	-10.00	82.59	45.05	-1.43	80.09	46.15	0.00	81.76
applu	67.04	-69.39	66.28	66.48	18.37	70.21	87.43	17.35	91.20
vpr	28.57	0.00	28.21	31.25	-19.08	36.65	38.84	-19.85	46.82
mesa	-5.71	-4.76	3.34	-2.86	4.76	6.59	-5.71	4.76	6.71
art	10.32	5.46	54.07	10.43	-5.77	55.30	11.53	-6.10	60.25
mcf	-0.37	-0.03	9.40	-0.27	0.49	9.77	-0.94	0.51	9.91
equake	63.13	-15.89	74.12	63.37	12.15	76.08	68.74	11.21	86.31
ammp	-0.22	1.09	12.71	-0.22	-2.19	12.30	-0.44	-2.19	13.14
parser	25.93	-5.58	78.06	26.33	-1.99	78.92	26.13	-1.79	79.71
gap	48.28	-9.33	96.26	48.28	0.00	96.23	48.28	0.00	98.61
bzip2	7.79	-6.95	36.38	7.41	1.05	34.52	8.17	0.84	40.17
twolf	-0.22	0.34	4.73	-0.11	-1.46	5.30	-0.33	-1.34	6.58
apsi	40.91	-14.29	64.45	40.91	0.00	64.50	40.91	0.00	65.28
sphinx	15.66	1.47	14.65	17.31	-3.91	14.99	17.03	-3.84	16.77
average	28.37	-9.39	48.62	26.80	0.64	50.05	29.34	0.55	54.34

Table 3. Coverage and accuracy of the other push schemes

cache misses and can interact with the prefetching and pushing techniques. This section provides results of combining GRP, evict-me, and data pushing.

4.1 Performance

For our experiments, we use the same system configurations as described in Section 3.2 except that we change the L1 cache line size to 32 bytes and make it 4-way set-associative. The Level 1 cache size does not change. We change the L1 cache line size so that we can examine the performance impact when the cache line sizes of the two levels of caches differ. We merge the Fortran benchmarks used in two previous papers [35, 34]. We do not include C benchmarks from SPEC CPU2000 because our compiler currently does not support dependence testing in C code very well.

Figure 1 compares the performance of GRP, evict-me (EM), and their combination. Evict-me is turned on for both levels of cache. GRP and Push use LRU and MRU placement policies respectively. Evict-me does not offer much performance improvement over the base, although there are no degradations. It improves *mgrid* and *arc2d* by about 4% and boosts overall performance by 1.5% on average across all benchmarks. GRP, which tolerates most L2 misses for these Fortran benchmarks, improves performance by 30.5% on average. Combining GRP and Evict-me adds an additional 1.7% and the Push scheme adds an extra 2.7%. The combination of GRP, Push, and evict-me boosts the base performance by 36.5% on average, adding an additional 6% over GRP. The performance gain mostly arises from *swim*, *jacobi*, and *vpenta*, with improvements over GRP by 13.2%, 14.9%, and 10.2%, respectively. For all but *vpenta*, combining the three techniques beats any single one or two combined. For *vpenta*, GRP/Push/EM is negligibly worse than GRP/EM because of the slight degradation of GRP/Push.

4.2 Cache Pollution

We use a pseudo direct-mapped structure to measure the pollution caused by L1 pushes. The structure uses the same line size as the L1 cache and its total number of lines equals the number of sets of the L1 data cache. When a pushed line evicts a cache line, we record the evicted line's address in the pseudo structure. On a demand L1 miss, we check if it hits the structure. If so, we consider the previously pushed line as having polluted the cache. Figure 2 shows the normalized L1 pollution caused by pushed lines of GRP/Push and GRP/Push/EM. Above each set of bars is the pseudo structure hit rate with GRP/Push, which we use as a metric of cache pollution. The overall pollution caused by pushed lines is small. Evict-me reduces this pollution by over a half. For three benchmarks, *apsi*, *jacobi*, and *tomcatv*, evict-me eliminates almost all pollution.

4.3 Discussion

The evict-me cache shows less performance improvement here than reported in the previous paper [35]. We attribute this to three reasons. First, the ISAs are different. The previous work targeted the SPARC V8 ISA, while here we use the Alpha. Different back-end optimizations have an impact on data layout and data access patterns, which affect cache replacement. Second, we use two different simulators, URSIM and SimpleScalar. URSIM is designed to model multi-processor systems. It implements strict cache inclusion while SimpleScalar does not. This choice has a significant impact on cache replacement decisions. In an inclusion system, an L2 replacement will invalidate the corresponding L1 cache lines. On an L1 miss, an invalid line, if it exists, will be evicted before an evict-me line or an LRU line. Third, cache ports are not modeled in SimpleScalar, which makes the caches in SimpleScarlar very aggressive and assumes infinite parallelism in cache accesses. It will be interesting to see how evict-me performs in SimpleScalar when we implement a stricter cache model. We envision the stricter model will make cache performance more critical and create more improvement space for GRP and Evict-me.

5 Related Work

Most pertinent to this work are two previous papers. First, guided region prefetching, proposed by Wang et al. [34], uses compiler hints to guide an aggressive region prefetching engine. We extend GRP by adding a push scheme to push the prefetched lines from L2 to L1 to hide L1 latencies. Second, Wang et al. [35] propose evict-caching which uses compiler hints to assist cache replacement. We combine evict-me caching with GRP to investigate the impact of cache replacement on the effectiveness of prefetching. These work including the current paper distinguishes themselves in their hardware/software cooperative approach. Below we first investigate related prefetching work and then justify our unique contribution on combining data prefetching and cache replacement.

Enormous amount of previous work on prefetching typically focuses on either software prefetching or hardware prefetching. Software prefetching relies on non-binding prefetch instructions that bring the indicated block of memory into the cache, much like a load instruction. Because the compiler only inserts prefetches for known (or very likely) loads, software prefetch accuracy is typically high. However, the performance of software prefetching is typically low due to two key challenges in data prefetching to the compiler: *selection* and *scheduling*.

Accurate compile-time identification (selection) of the loads that will cause cache misses at runtime is complex, requiring both knowledge of hardware parameters (cache block size, capacity, and associativity) and sophisticated code analysis (e.g., to determine the volume of other data

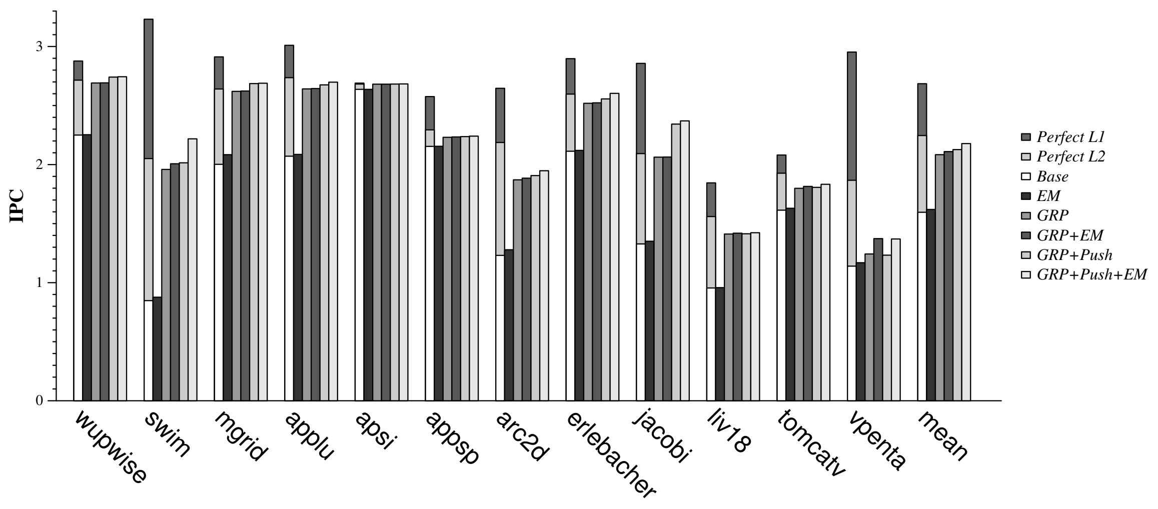

Figure 1. Evict-me and GRP

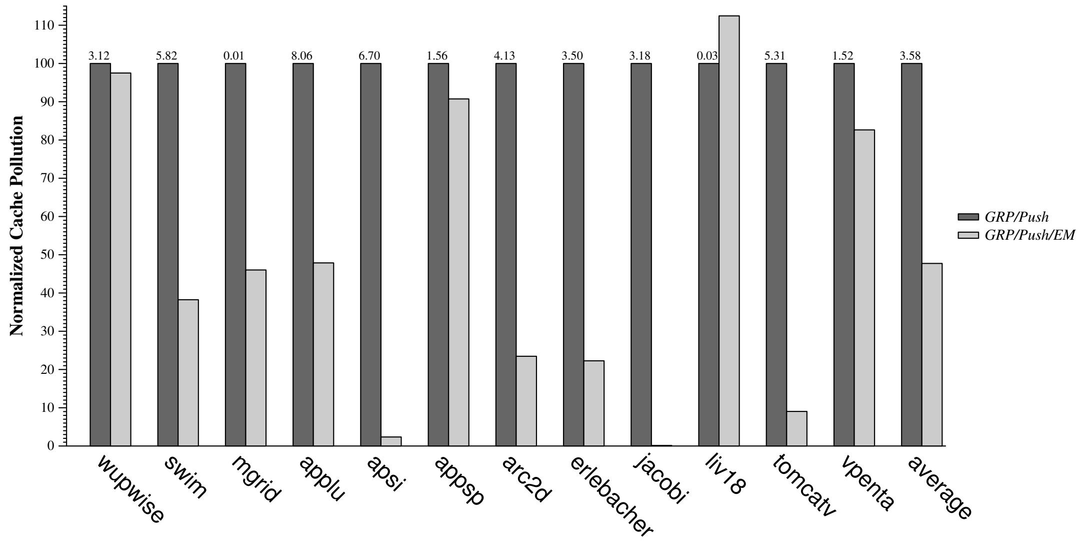

Figure 2. Cache pollution

accessed between references to a particular block) [5, 11, 23, 36]. The compiler also faces the difficult challenge of issuing the prefetches sufficiently early to hide the memory latency, but not so early that useful data are needlessly evicted. While this constraint is not significant for arrays [4, 23], it limits compiler-based greedy pointer prefetching [3, 21, 27]. Jump pointers bypass this limitation by identifying records several links ahead in the structure, but require much more sophisticated analysis, dynamic updates, and the addition of a jump pointer to each object [3, 21, 27]. Other approaches prefetch pointer arguments at call sites [20], and decouple prefetches from the main program using a separate thread context [9, 16, 22].

The converse approach is hardware-only prefetching, in which the hardware predicts prefetch addresses by observing a program's runtime behaviors [1, 6, 25, 10, 14, 26, 13, 30, 31, 37]. Since prefetches do not not incur overhead in the processor itself, the hardware need not be as selective about issuing prefetch operations. However, unlike the compiler, the hardware has no direct knowledge of future memory references; the key challenge in hardware-based prefetching is determining a reasonable set of predicted addresses to use as prefetch targets. Hardware prefetching thus suffers relative to software prefetching in both accuracy (because the predictions may be wrong) and coverage (because some addresses may require the compiler's scope to predict).

GRP combines the advantages of both software and hardware prefetching in a scheme that is simple yet effective. It conveys sophisticated compiler analysis by associating a range of hints with loads, which an aggressive, simple, and general hardware prefetcher uses only when necessary. Thus, the pertinent compiler analysis is communicated to the hardware without requiring extensive static lookahead, software guarantees, or high instruction overhead.

The limited previous work use an cooperative approach has either exploited prefetching for restricted classes of access patterns, or provided an interface that is overly general and complex. On the conservative side, Gornish and Veidenbaum [12] let software select the number of contiguous blocks to prefetch upon a miss, whereas Chen and Baer [7, 8] use the compiler to supply address and stride information to augment a reference prediction table. Skeppstedt and Dubois use a trap handler to trigger prefetching using similar information [29]. In the Power 5 prefetching design, a sequence of L1 misses trigger aggressive prefetching of up to 8 streams into the L1 and further ahead into the L2. It paces itself to stay one line ahead in the L1 and 10 in the L2. In addition, the software may indicate the length and initiating address of a stream [32, 24]. Karlsson et al. [15] use *prefetch arrays* to enable a hardware engine to perform a generalized variant of greedy and jump-pointer prefetching. Zhang and Torrellas [38] use the compiler to mark blocks in memory as belonging to contiguous spatially local regions or containing indirection pointers. Their scheme requires additional bits in main memory and significant support in the memory controller. Finally, fully programmable prefetch engines provide flexibility but require significant memory system support and have not yet demonkstrated that the required compiler support is realistic [31, 33, 37].

Although the previous work on region prefetching alleviates the pressure on cache replacement by prefetching data into the LRU lines [19], it is still lack of a though investigation how cache replacement policy interacts with data prefetching. One close report in this area is by Lai et al. [17] who use a hardware history table to predict when a cache block is dead and which block to prefetch to replace the dead one. Their technique is hardware-only and thus does not benefit from static compiler knowledge.

6 Conclusion

In this paper, we evaluate the synergy among GRP, data pushing, and evict-me. We show that both the push scheme and evict-me bring an additional performance improvement over GRP. The three techniques together add an additional 6% over GRP. The evict-me cache does not perform as well in SimpleScalar as in URSIM. We attribute this to differences in the compiler, in the accuracy of the L1 cache models, and that cache inclusion is not enforced in SimpleScalar. However, it is worth further investigation to examine where exactly the gap arises. Even in the aggressive cache model of SimpleScalar, we observe that evict-me is very effective at reducing cache pollution caused by prefetched or pushed lines. It eliminates half the L1 cache pollution from the data push scheme. This result suggests our cooperative techniques have potential to interact well to improve memory system performance.

References

[1] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, 2001.

[2] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[3] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compiler Techniques*, pages 280–291, Barcelona, Spain, Sept. 2001.

[4] B. Cahoon and K. S. McKinley. Simple and effective array prefetching for Java. In *ACM Java Grande*, pages 86–95, Seattle, WA, Nov. 2002.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 40–52, Santa Clara, CA, Apr. 1991.

[6] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, MA, Oct. 1992.

[7] T. Chen and J. Baer. Effective hardware based data prefetching. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[8] T.-F. Chen. An effective programmable prefetch engine for high-performance processors. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 237–242, Ann Arbor, Michigan, Nov. 1995.

[9] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 14–25, June 2001.

[10] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 56–63, St Charles, IL, 1993.

[11] 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.

[12] E. H. Gornish and A. V. Veidenbaum. An integrated hardware/software scheme for shared-memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages 281–284, St Charles, IL, 1994.

[13] C. J. Hughes and S. Adve. Memory-side prefetching for linked data structures. Technical Report UIUCDCS-R-2001-2221, University of Illinios, Urbana Champagne, May 2001.

[14] 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.

[15] M. Karlsson, F. Dahlgren, and P. Sternstrom. A prefetching technique for irregular accesses to linked data structures. In *Sixth International Symposium on High Performance Computer Architecture*, pages 206–217, Toulouse, France, Jan. 2000.

[16] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, Oct. 2002.

[17] A.-C. 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, July 2001.

[18] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recoginition system. In *IEEE Transactions on Acoustics, Speech and Signal Precessing*, volume 38(1), pages 35–44, 1990.

[19] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301–312, Jan 2001.

[20] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microachitecture*, pages 231–236, Nov. 1995.

[21] C. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, Oct. 1996.

[22] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 40 – 51, Goteborg, Swenden, June 30 - July 4 2001.

[23] 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.

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

[25] 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.

[26] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceeding of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, 1998.

[27] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 111–121, Atlanta, GA, May 1999.

[28] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Barcelona, Spain, Sept. 2001.

[29] J. Skeppstedt and M. Dubois. Hybrid compiler/hardware prefetching for multiprocessors using low-overhead cache

miss traps. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 298–307, Bloomington, IL, Aug. 1997.

[30] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[31] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 171–182, May 2002.

[32] J. M. . Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 40(1), 2002.

[33] S. P. VanderWiel and D. J. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of International Conference on Computer Design*, pages 372–377, Austin, TX, 1999.

[34] 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.

[35] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *The 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–208, Charlottesville, Virginia, Sept. 2002.

[36] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 210–221, Berlin, Germany, June 2002.

[37] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 176–186, May 2000.

[38] Z. Zhang and T. Torrellas. Speeding up irregular applications in shared memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 1–19, Santa Margherita Ligure, Italy, June 1995.