Extending Data Prefetching to Cope with Context Switch Misses

Hanyu Cui*
Qualcomm
hanyuc@qualcomm.com

Suleyman Sair*
Intel
Suleyman.Sair@intel.com

Abstract— **Among the various costs of a context switch, its impact on the performance of L2 caches is the most significant because of the resulting high miss penalty. To reduce the impact of frequent context switches, we propose restoring a program's locality by prefetching into the L2 cache the data a program was using before it was swapped out. A Global History List is used to record a process' L2 read accesses in LRU order. These accesses are saved along with the process' context when the process is swapped out and loaded to guide prefetching when it is swapped in. We also propose a feedback mechanism that greatly reduces memory traffic incurred by our prefetching scheme. Experiments show significant speedup over baseline architectures with and without traditional prefetching in the presence of frequent context switches.**

I. Introduction

Most time-shared operating systems use context switching since it leads to faster response times, higher throughput and fairness as well as higher utilization of system resources. However, context switches hurt cache performance as cache state belonging to different programs compete for cache space and replace each other's lines. Since most of these interfering programs do not share any data or instructions, the cache content built up by the swapped out process is of no use to the new process. It would bring all its data and instructions into the cache from main memory. Given the speed gap between memory and on chip caches, this creates a significant bottleneck.

In this paper, we propose restoring a program's cache state via prefetching to mitigate the impact of context switches. Prefetching has several advantages over previous schemes that target context switches: (1) Prefetching frees the OS to focus on other scheduling criteria (responsiveness, fairness etc.) by removing restrictions such as minimum time quantum limits or increased the priority of the current thread; (2) In cases where frequent context switches are inevitable, e.g. in the presence of network streaming, multimedia processing or inter-process piping, OS based techniques do not perform well; (3) Cache partitioning is fundamentally geared towards a different problem (resolving conflicts among simultaneously running threads in a CMP) and is ineffective when there are more threads than partitions.

To restore a program's locality, we use a list to save the blocks accessed by the program when it is running. At the time it is swapped out, the addresses of these blocks are saved with the context of the program. The next time the program is swapped in, these addresses are loaded into a queue to guide prefetching. This effectively restores the cache contents of the program before it was swapped out. Compared with prefetching techniques which rely on arithmetic patterns in address streams, our technique can capture irregular accesses. And unlike common context-based prefetchers (e.g. Markov [1]) which are limited by predictor table size and are themselves subject to loss of content during context switches, our technique stores the contents of the prefetcher along with the program state so that it can be restored upon being swapped in.

*This research was conducted while the authors were with NC State University.

The contributions of this paper include: (1) tracking which blocks are accessed before a program is swapped out using an LRU-ordered doubly-linked list; (2) saving the blocks along with a process' state and prefetching them the next time the process is swapped in; (3) presenting a placement policy which is tailored to increase the lifetime of prefetches in the cache; (4) designing a feedback mechanism which significantly reduces memory traffic incurred by prefetching; and (5) attaining 36% average speedup over no prefetching, and 11% and 24% average speedup in the presence of other prefetchers.

II. Related Work

There are many prefetching schemes in the literature. An early example of a prefetching architecture is Nextline Prefetching (NLP) by Smith [2], where each cache block was tagged with a bit indicating a prefetch should be issued. Using this bit, when a prefetched block is accessed by the program, a prefetch of the next sequential block is triggered. A stride prefetcher [3] keeps track of the difference between the last address of a load and the address before that, which is called the stride. The prefetcher speculates that the new address seen by the load will be the sum of the last address value and the stride. In Markov prefetchers [1], each missing address would index into a Markov prediction table to provide the set of cache addresses that have followed this address before. Prefetches are issued for these address under the heuristic that a miss is likely be followed by the same set of misses. While these techniques can be useful to eliminate context switch related misses, their address predictor tables are also subject to clobbering during the execution of another process. Thus, when a swapped out process is brought back in, its address predictor data needs to be rebuilt before prefetching can be useful.

The closest prior work to ours is that of Nesbit et al. [4] where a Global History Buffer (GHB) is proposed for holding the most recent miss addresses in FIFO order for subsequent prefetching. The GHB has two advantages over traditional predictor tables: it eliminates stale prediction data, and it maintains a complete picture of cache miss history. The GHB is similar to our approach in that both

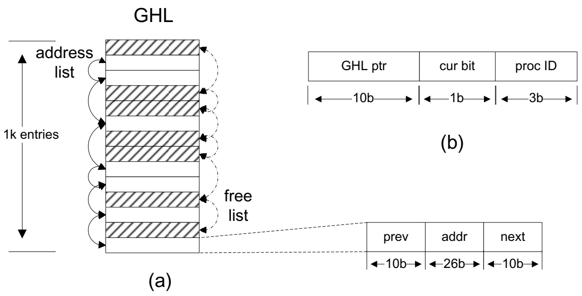


Fig. 1. (a) The Global History List, shown here with 1K entries. Blank entries are containing valid block address and linked together to form the address list. Shaded entries are unused entries and form the free list. As shown in the bottom, each GHL entry has three fields. Prev and next point to the previous and next entry in its own list respectively. (b) Additional information kept in each cache line. "GHL ptr" is a 10-bit pointer pointing to the corresponding GHL entry. "cur bit" indicates whether the cache is brought in by the current process. "proc ID" indicates by which process the block is brought in.

are trying to record the global history. However, they have three fundamental distinctions: (1) The GHB only records misses while our scheme records all read accesses; (2) Our scheme tries to remove duplicates and compact the history while the GHB does not because doing so would break how it operates; (3) We save the access history along with the state of the process while it is being swapped out so that it can later be restored. As a result, our scheme is more suited to warming up cache state after a context switch.

III. Context Prefetching

Frequent context switches are inevitable in certain cases. To measure the frequency, we used SystemTap [5] to collect the context switch trace from a real machine. The machine is a single core Pentium 4 3.0 GHz machine running Linux Kernel 2.6.17. When the system is running `scp` and `gcc`, processes can run no more than 600 μs before a context switch occurs. Our experiments show that such frequent context switches have a significant impact on the performance of the L2 cache and the entire system.

Given the pressing need to mitigate the impact of context switches, we propose restoring the L2 cache content via prefetching. Our prefetching infrastructure centers around a *Global History List* (GHL), along with certain block-specific information kept in each cache line.

The GHL is a buffer organized into two doubly-linked lists: the *Address List* and the *Free List*, which are shown in Figure 1 (a). Each GHL entry has three fields: an L2 cache physical block address and two pointers. The pointers point to the previous entry and next entry respectively, either in the address list or the free list. The address list is used to record all L2 read accesses, while the free list holds the unused entries. To record an L2 access, one entry is taken from the free list and inserted at the tail of the address list. To reclaim an entry, we remove it from the address list and insert it back into the free list. The pointers in the affected entries are changed accordingly in these operations.

The address list is maintained in LRU order meaning new block addresses are inserted at the tail. We record the accesses on a per-process basis. When the process is swapped out, the block addresses in the address list are saved in a dedicated region in main memory. The next time this process is swapped in, the block addresses will be loaded into the GHL to guide prefetching. We issue prefetches starting from the MRU entry to bring in the most recently used data items into the L2 cache first.

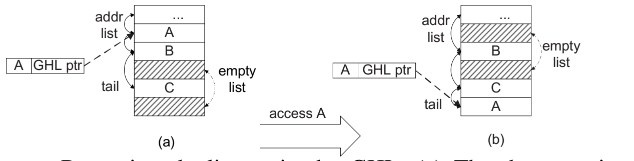


Fig. 2. Removing duplicates in the GHL. (a) The three entries closest to the tail of the address list contain C, B and A respectively. The GHL pointer in the cache line that contains block A is pointing (the dashed line) to the corresponding entry in the address list. (b) After an access to block A, the old entry that contains A is reclaimed (becomes shaded) and a new entry is allocated for A at the tail of the address list. The GHL pointer then points to the new entry.

A pointer, as shown in the "GHL ptr" field in Figure 1 (b), is kept in each cache line to help reduce the number of duplicates in the address list. The pointer points to the GHL entry that corresponds to the address of the block contained in the cache line. As shown in Figure 2, if a new access hits at this line, it must have the same block address as the entry identified by the pointer. Thus, this duplicate entry can be reclaimed. The doubly-linked structure of the GHL facilitates easy removal of duplicate entries.

To reduce the on-chip area overhead, we adopt a two-level GHL scheme with a small on-chip component and a larger off-chip part. The *on-chip GHL* stores the most recently accessed addresses. The off-chip part, referred to as the *off-chip GHL*, is a circular buffer in a dedicated region of the main memory. When the on-chip GHL is full, the oldest entry will be moved to the off-chip one and the GHL pointer in the corresponding L2 cache line will be invalidated. If the off-chip GHL is full, the oldest entry is overwritten.

Our experiments show that a 16K-entry GHL where 1K entries are on chip and 15K are off chip is a good trade-off between performance and area. For a block size of 64 bytes and a 32-bit memory address space, 26 bits are required to store a block address. With a capacity of 1K on-chip entries, each pointer in the GHL takes up 10 bits. All together, the on-chip size is less than 100KB. The GHL and related components are not on the critical path. They only communicate with the L2 cache and the main memory bus

A. Placement Policy

Since thousands of blocks are being prefetched, it is very important to not replace useful blocks, which include two categories: (a) blocks brought during the current interval, including demand and prefetched blocks; (b) blocks brought in during the previous execution of the current process. The first category is referred to as *current blocks* and the second category is referred to as *survivor blocks*.

To identify current blocks, we keep a *current bit* field in each cache line, as shown in Figure 1 (b). This bit is cleared when a new process is swapped in and set when a block is placed in the line by a prefetch or a demand access. When a block is prefetched and there are no empty lines in a set, we first try to replace a line whose current bit is zero. If no such line can be found, we replace the LRU line.

B. Operations

GHL-prefetching includes maintaining the access history during execution, saving the address list upon a context switch, and after being swapped in, loading the address list and issuing prefetches.

Recording accesses: Whenever there is an L2 read access, an entry is allocated in the on-chip GHL and inserted at the tail of the address list. If this insertion results in duplication, the older entry will be removed. Note that when a cache line is replaced, we do not reclaim the address list entry corresponding to the replaced line. Therefore, it is possible that an address is still in the on-chip GHL but the corresponding block has been replaced in the cache. In this case, the pointer to the on-chip GHL entry is lost and there is no way to reclaim that entry even if it is a duplicate. Our experiments show that this does not result in a noticeable waste of entries. When there is an L2 access but no entries in the free list, we move the oldest entry in the address list to the off-chip GHL. The oldest entry in the off-chip GHL will be overwritten when it becomes full.

Saving the address list: When a process is swapped out, besides saving its context, its address list in the on-chip GHL is merged with the off-chip GHL. As each entry is saved, we add it to the free list in preparation for the next process. Note that only the block address field in each address list entry is saved to memory. The linked-list pointers are installed when the list is repopulated upon swapping the process in.

Loading the address list and issuing prefetches: When a process is swapped in later, the addresses saved in the off-chip GHL will be loaded into a prefetch buffer to guide prefetching. Since the buffer has only 1K entries, prefetching will begin after it has been filled or all saved addresses have been loaded. A single prefetch is issued every cycle from the prefetch buffer. The next 1K will be loaded after prefetches have been issued for the current ones. The first 1K addresses will also be inserted at the head (LRU entry) of the address list of the on-chip GHL, until it becomes full. When a prefetch is issued for any of these 1K entries, the "GHL ptr" in the destination cache line is changed to point to this entry. This enables duplicate removal for the 1K MRU addresses. Unlike a demand access, issuing a prefetch does not move the entry to the tail of the address list. We consider the time it takes to save and repopulate the first 1K entries of the address list to be part of the context switch overhead. The latter parts of the off-chip GHL are brought in as bandwidth permits while the process is running.

The OS needs to have a few modifications in order to support these operations: (1) The OS needs to activate and deactivate GHL for a process. It also needs to tell GHL when a context switch happens and wait for GHL to finish saving and loading addresses. (2) The off-chip GHL resides in a dedicated region in the main memory. The addresses in the on-chip GHL also need to be saved in this region when the process is swapped out. Hence, the OS needs to set aside a region in the kernel space for each process. When the OS activates the GHL or a context switch happens, it also updates a pair of dedica ted registers with the beginning and ending addresses of the region for the current process. GHL will only access the region indicated by these two registers. (3) GHL prefetches physical memory blocks into the L2 cache. However, some types of memory pages are uncacheable and cannot be placed in the L2 cache. To make sure blocks in these pages are not prefetched, GHL records an access only if the accessed block can be placed in the L2 cache. This can be easily known at the time of the access. Another problem caused by uncacheable pages is that this attribute can be changed dynamically and blocks already in the caches and GHLs could become uncacheable. When this happens, the OS needs to flush the caches and the on-chip/off-chip GHLs. Since such changes are rare in real systems except in system initialization stage, they do not cause any noticeable performance degradation to GHL-prefetching.

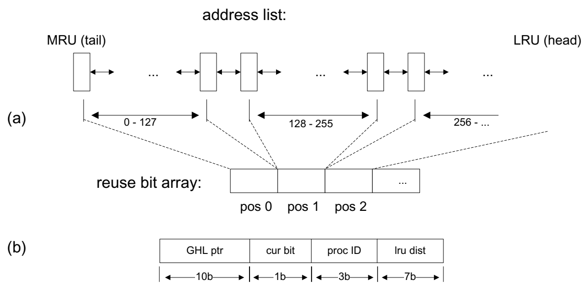


Fig. 3. (a) Mapping between the reuse bit array and the address list, e.g. position 0 in the reuse bit array corresponds to entry 0 through 127 in the address list. (b) In addition to the fields in Figure 1 (b), a "lru dist" field is added to each line, which contains the corresponding position in the reuse bit array.

C. Feedback Mechanism

We found that depending on the behavior of the program, the percentage of useful prefetches varies and sometimes is low enough to cause significant waste of memory bandwidth. Our experiments reveal that a certain cluster of prefetches would be used, followed by a series of wasteful prefetches. Furthermore, this pattern exhibits temporal locality across different executions of the same process. Hence, we use a *reuse bit array* to record which prefetched blocks are used by the program. Each position in this reuse bit array corresponds to a GHL entry allocated a certain number accesses ago (e.g. the LRU order). Since the address list is also in LRU order, the reuse bit array can be mapped to the entries in the address list and used to track which addresses in the address list are used by the program. If the access pattern of a program remains fairly stable across context switches, prefetching only the addresses that are used eliminates most of the useless prefetches. Our experiments show that this assumption holds for most of the benchmarks we studied.

Our experiments also reveal that tracking the utilization of each prefetched block is too fine-grained. Thus, a single bit is used to identify the reuse of blocks in a region of

GHL entries. More precisely, a single bit is mapped to 128 consecutive entries in the address list of the GHL in our design, as shown in Figure 3 (a). For a GHL with 16K entries, the reuse bit array needs to have 128 bits. To record which blocks are reused, we store the reuse bit array position in each cache line, as shown in the “lru dist” field in Figure 3 (b). A context prefetch can only be issued if the corresponding bit in the reuse bit array is set. When a prefetch issued, the position of the corresponding bit is kept in the “lru dist” field in the destination cache line. When a demand access hits in this cache line, the bit indicated by “lru dist” is set to indicate that this region has been shown to be useful. To eliminate the ambiguity between the lookup and update operations in the reuse bit array, two reuse bit arrays are kept. One records the reuse of prefetches during the current interval. The other one contains the reuse pattern of the previous execution and is used to issue prefetches for the current run. If a block in the cache is not brought in by a GHL prefetch, its “lru dist” field will be set to zero.

One issue with the feedback mechanism is that once we stop prefetching blocks in certain address list region, we can not detect should they become useful in the future. Thus, we issue a prefetch for the first block in each region regardless of its reuse bit value. As a result, if the prefetched block is reused, the corresponding bit in the reuse bit array would be set and cause the entire region to be prefetched. This would result in a minor waste of memory bandwidth, but ensures that we do not ban a certain range in the address list forever.

D. GHL-NLP Hybrid Scheme

Recognizing that GHL-prefetching and NLP are complementary, we designed a hybrid scheme that tries to take advantage of the strengths of both schemes by adaptively alternating between the two prefetchers. Since GHL-prefetching already has a feedback mechanism in place, we use this feedback information to choose a prefetching scheme after a context switch. When more than 50% of the bits of the reuse bit array are set, only GHL will be used for prefetching. When fewer than 50% of the bits are set, this is a good indication that GHL is not being effectively utilized. In this case, we activate NLP and not use GHL-prefetching. However, GHL-prefetching will continue to record accesses and one prefetch will still be issued for each region indicated by the reuse bit array, as mentioned in the previous section. Experiment results show that by taking advantage of the reuse bit array, the better prefetcher can be selected in most cases.

IV. Methodology

Our simulator is based on the timing simulator in SimpleScalar 3.0 toolset [6] for the Alpha AXP ISA. Its configuration is shown in Table I. We extended the baseline simulator to model queuing at the various levels of the memory hierarchy.

We used the SimpleScalar cache simulator with similar configuration to our timing simulator to collect L2 read access traces for several of the SPEC’2K benchmarks. We use these traces to mimic the effects of a context switch on the cache hierarchy. At each context switch, for each address in the trace, we clear the valid bit of the corresponding L2 cache line. Following a context switch, we assume the interfering benchmark runs for 1M instructions (only part of which are loads). After this, the main application resumes execution. Since we do not have a scheduler, we trigger context switches at a fixed period. We also tried random context switch intervals and found that there was not a noticeable difference as long as the mean of the random intervals was the same as the fixed period.

Parameter	Value
Fetch/Decode/Retire width	4 instructions
RUU size	128
Load/store queue size	64
Function units	4 intALU, 1 int mul/div
Branch Pred.	256-1K 2-level predictor
L1 D-cache/I-cache	16KB, 2-way set assoc., 64 byte lines, 2 Cycle hit latency
L2 cache	2MB, 16-way set assoc., 64 byte lines, 18 Cycle hit latency, 350 Cycle miss latency
L2/MEM bus bandwidth	8 Bytes/cycle

TABLE I
ARCHITECTURAL CONFIGURATION.

TABLE II
BASELINE BANDWIDTH UTILIZATION.

Benchmark	Utilization(%)	Benchmark	Utilization(%)
ammp	8.85	gzip	8.11
applu	23.50	lucas	13.63
apsi	2.96	mcf	13.32
art	29.01	mesa	4.15
bzip2	8.88	mgrid	11.70
crafty	2.87	parser	4.77
eon	1.59	perl	3.95
equake	11.53	sixtrack	2.09
facerec	5.71	swim	17.33
fma3d	6.81	twolf	6.63
galgel	14.54	vortex	4.11
gap	3.15	vpr	7.93
gcc	4.18	wupwise	7.41
Average (%): 7.79			

We evaluate our scheme with all benchmarks in the SPEC’2K benchmark suite. Table II shows the baseline memory bus utilization of the benchmarks. It can be seen that all benchmarks except `applu` and `art` use less than 20% of the available bandwidth. This leaves ample idle bandwidth for the prefetchers to consume.

For each benchmark we use the reference input set and simulate a single simulation point obtained with SimPoint [7]. Each simulation alternates between the simulated benchmark and an interfering benchmark until the main application is executed for 500 million instructions. Unless noted otherwise, the results presented in the next section are the average across all benchmarks and are collected with the parameters shown in Table III.

V. Evaluation

In this section we evaluate the performance of GHL-prefetching methods outlined in Section III. We compare

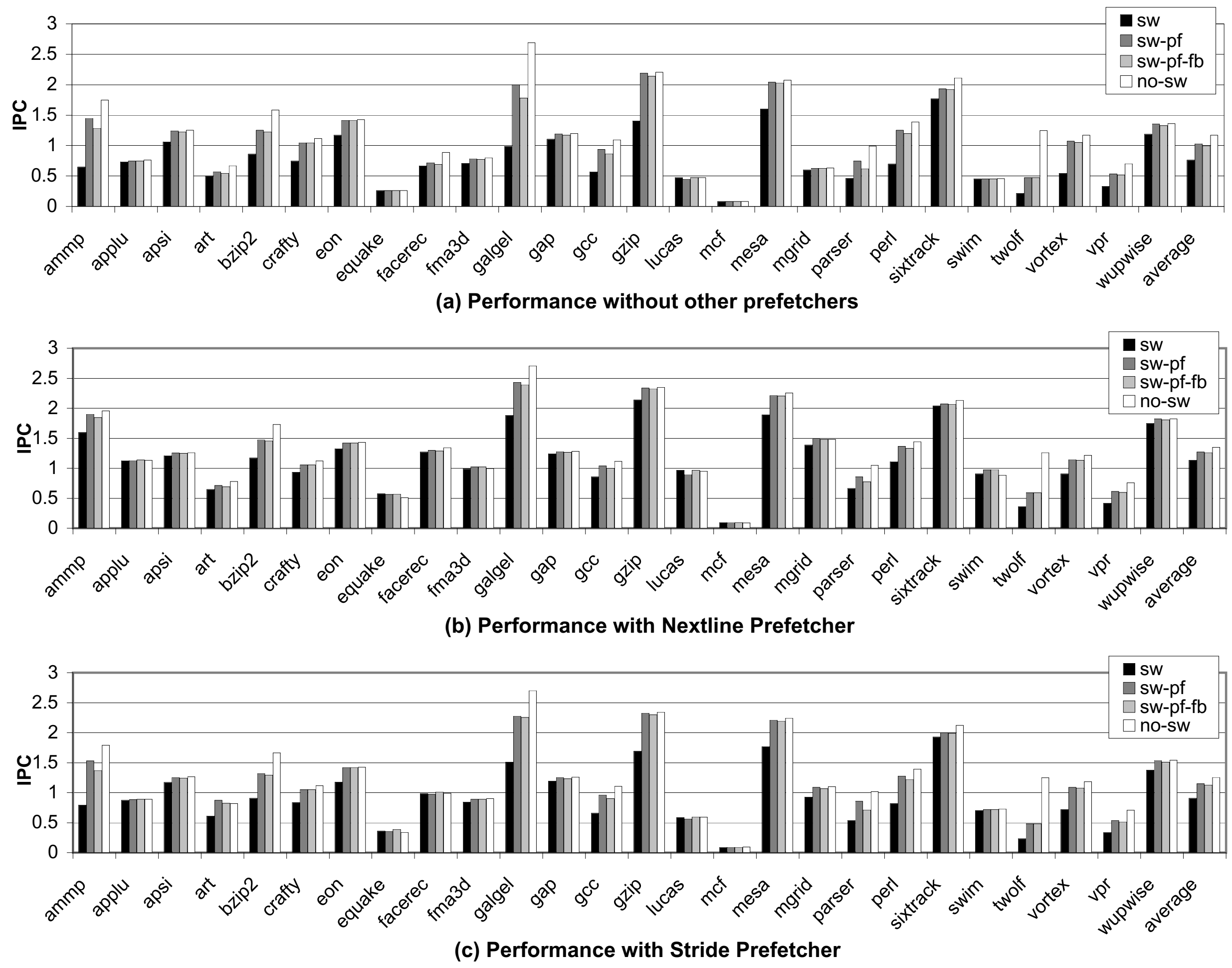


Fig. 4. Performance with and without GHL-prefetching. Four cases are compared in each graph: context switch is present (*sw*), context switch with GHL-prefetching (*sw-pf*), context switch with GHL-prefetching and feedback (*sw-pf-fb*), and no context switch (*no-sw*).

Parameter	Value
GHL size	16K entries, 1K on-chip, max 15K off-chip
Reuse bit array	128 entries
Interfering benchmark	art
Context switch period	1M cycles
Interfering benchmark duration	1M instructions
Nextline Prefetcher	Prefetch degree is 4
Stride Prefetcher	PC-indexed, direct map, 4k entries, prefetch degree is 4

TABLE III
SIMULATION PARAMETERS.

GHL-prefetching to a baseline with no prefetching, as well as assuming a Nextline (NLP) and a Stride prefetcher in the memory hierarchy. The best case scenario compared against is the case with no context switches.

Figure 4 (a) depicts the case where the baseline architecture does not prefetch. In this scenario, GHL-prefetching results in a 36% average speedup across the entire SPEC'2K suite in the presence of frequent context switches. Most benchmarks benefit from context prefetching and more than half the benchmarks have significant speedup over no prefetching. It is also important that there is no slow down in any benchmarks. For some benchmarks, like `gzip`, our technique restores almost the entire performance loss caused by context switches. Some benchmarks, like `twolf` and `galgel`, suffer seriously from frequent context switches. Our scheme boosts their performance significantly, but not to the level of no context switches. After we further investigated their behavior, we found that these benchmarks have a large working set and require a longer GHL. For example, with a GHL size of 128K, `twolf`'s performance can be restored to within 5% of the performance without context switches. As expected, for benchmarks that do not suffer from context switches, e.g. `equake` and `lucas`, our technique does not help. It is also shown in the figure that our feedback mechanism causes a 3% IPC reduction, but later results in Figure 6 (a) will show that it is worthwhile for its ability to reduce memory bandwidth usage significantly.

Figure 4 (b) and Figure 4 (c) show the performance in the presence of a baseline NLP and a Stride Prefetcher respectively. Our scheme still brings 11%-24% respective speedup in these cases. This demonstrates that our scheme is orthogonal to these prefetching schemes and captures access patterns that other techniques might miss. It is interesting to notice that some benchmarks, e.g. equake and swim, perform better when both context switches and

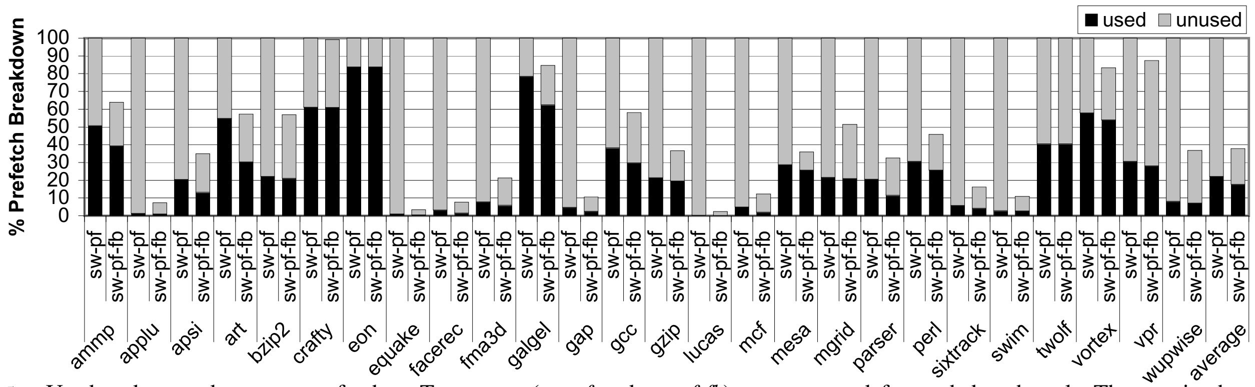


Fig. 5. Used and unused context prefetches. Two cases (*sw-pf* and *sw-pf-fb*) are presented for each benchmark. The y-axis shows the percentage relative to the number of context prefetches without feedback (*sw-pf*), i.e. prefetching with feedback issues fewer prefetches.

Nextline/Stride prefetching are present than when there is no context switching. We notice these benchmarks do not suffer from context switches, but are helped by an NLP or a Stride prefetcher. After further investigation, we discovered that this is a result of the interaction between NLP/Stride prefetching, our placement policy, and context switches. According to our prefetch placement policy, if there are no lines with the valid bit or the current bit cleared, we will replace the LRU line to store a prefetch. When there are no context switches, most lines are valid or current, and the majority of the prefetched lines can only go to the LRU line. If the prefetched block is not used soon, it is replaced by a subsequent demand access or another prefetch. However, when there are context switches, the current bit in conflicting cache line is cleared. Thus, after a context switch, most prefetches survive for a longer period in the cache bringing more benefit. For most benchmarks, the benefit gained in this way is much smaller than the impact incurred by a context switch. Another surprising observation is that NLP works better than stride prefetching for most benchmarks at the L2 level. This is because the PC-indexed stride predictor does not start issuing prefetches until it detects a stride and it can not capture global address regularities that NLP can exploit.

In Figure 5, GHL-prefetches are broken down into used and unused prefetches. The y-axis in the graph shows the percentage of total prefetches relative to the number of GHL-prefetches without feedback (*sw-pf*). It is shown that more than 70% of the issued GHL-prefetches are not used without the feedback mechanism. And it can also be clearly seen that our feedback mechanism significantly reduces the number of useless prefetches at the cost of a small reduction in the number of useful prefetches. The results also explain why some benchmarks, e.g., `applu`, `equake` and `lucas`, do not benefit from our scheme. The prefetches issued for these benchmarks are almost never used. The useless prefetches for some benchmarks, like `galgel` and `twolf`, are not reduced by the feedback mechanism at all. What is even worse is that it reduces only the useful prefetches for `galgel`. When we examined the reuse patterns of prefetches for these benchmarks, we found that they do not cluster as most other benchmarks do, thus rendering the grouping of prefetches ineffective. Furthermore, they do not exhibit the reuse pattern learned during the previous execution. These could have

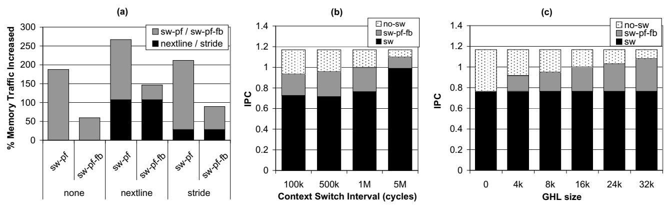


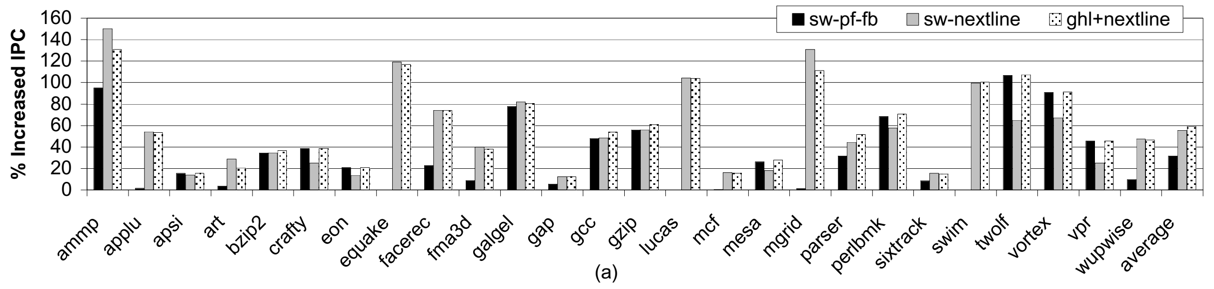
Fig. 6. (a) Percentage memory traffic increased in the presence of different prefetching schemes. Results are presented for GHL-prefetching only (*none*), with NLP (*nextline*) and with a Stride prefetcher (*stride*). For each of these cases, results with and without feedback are shown. (b) Performance with different context switch intervals. (c) Performance with different GHL sizes.

been captured by more sophisticated schemes, but the added complexity does not justify the meager gains they provide.

Figure 6 (a) shows the extra memory traffic incurred by different prefetching schemes relative to the no prefetching case where context switches are present (*sw*). The gray segment of each bar is the memory traffic caused by GHL-prefetching. The dark segment of each bar is the traffic resulting from NLP or Stride prefetcher. Although our scheme results in a 36% speedup (Figure 4), it also incurs 185% more memory traffic. When we use the proposed feedback mechanism, it brings the bandwidth overhead down to 60% with a slightly lower speedup of 31% (Figure 4). While discussing Figure 4, we had mentioned that NLP outperforms stride prefetching. Figure 6 (a) depicts the cost at which this speedup is obtained. The Stride prefetcher generates a quarter of the extra memory traffic when compared to NLP.

Figure 6 (b) shows the impact of context switch intervals. We can see that longer intervals (5M cycles) tend to have a smaller impact on a program's performance. A 1M-cycle context switch interval tends to degrade performance a lot while 100K-cycle and 500K-cycle intervals are similar to the 1M interval. In contrast, the impact of context switch intervals in the presence of our scheme is much less. The figure demonstrates that our technique removes at least half the impact of context switches on average and as shown earlier, we maintain the performance close to when no context switches are present across a wide range of intervals.

Figure 6 (c) shows the effect of changing the length of the GHL. In the chart, it is shown that a 4K GHL already does quite well. On average, it results in a 20% speedup. On



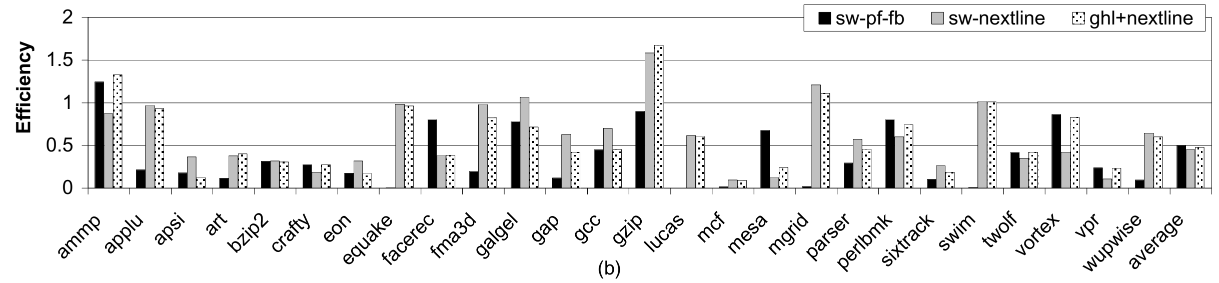


Fig. 7. Comparing GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and the GHL-NLP hybrid scheme (*ghl+nextline*): (a) Speedup. (b) Prefetch efficiency.

average, a 16K GHL only brings 11% more speedup when compared to the 4K GHL. We retain a 16K GHL because it does make a clear difference for some of the benchmarks.

Since GHL-prefetching and NLP are complementary, we also examine the hybrid scheme that adaptively selects between these two prefetchers, as described in S ection III-D. Figure 7 (a) examines the speedup of GHL-prefetching, NLP and GHL-NLP hybrid. It can be seen that GHL-NLP hybrid has equal or higher speedup than GHL-prefetching and NLP for most benchmarks, and its speedup is even higher than both for `parser`. For `ammp`, `art` and `mgrid`, the hybrid scheme has slightly lower speedup than NLP. This is because the feedback mechanism could not detect all the cases where GHL-prefetching is less effective than NLP. On average, GHL-NLP hybrid has the highest speedup, which is 5% higher than NLP's. Figure 7 (b) shows the efficiency of the three schemes. We define efficiency as the ratio of the percentage of increased IPC to the percentage of increased memory bandwidth. GHL-NLP hybrid has slightly higher efficiency than NLP and enjoys the highest speedup because it is able to select the better prefetching scheme in most cases.

In order to justify spending the area on our scheme rather than on a larger cache, we experimented with a 2.5MB L2 cache (*sw-2.5M$*) and a 3MB L2 cache (*sw-3M$*) and compare the performance with our feedback controlled GHL-prefetching scheme with a 2M L2 cache (*sw-pf-fb*). The results are shown for different interfering benchmarks in Figure 8. Our scheme uses less than 100KBytes for a 2MB cache with 64-byte block size but outperforms even a 3MB L2 cache for interfering benchmarks with intermediate to high impact. Unlike caches, which only try to keep a process' working set on chip for the duration when a process occupies the processor, our scheme tries to maintain the process' working set across context switches. This is achieved by spending some area on saving which data is needed rather

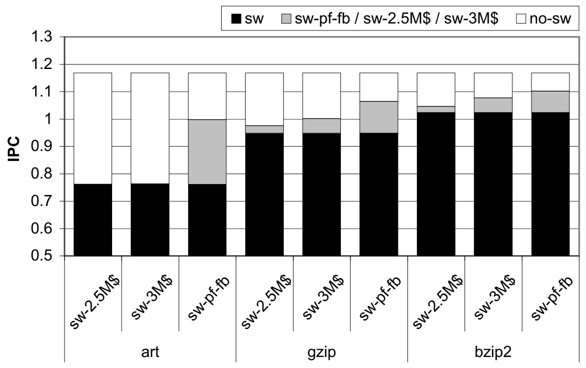


Fig. 8. Performance of our GHL-prefetching mechanism with a 2MB L2 cache compared to the performance of larger caches.

than saving the data itself. Since this information uses much less space than the data itself, it is possible to save and load it in much less time and use it to guide prefetching. As shown earlier, when there are multiple interfering benchmarks, more than half of the original application's blocks will not survive. Thus, the area spent on caches could be used much more efficiently by using it for context prefetching.

Besides using a GHL, we also explored other approaches that reduce context switch misses, such as saving the L2 cache tags. We investigated saving all the tags, or a certain number of MRU tags from each cache set. Figure 9 compares GHL-prefetching to saving the L2 tags. The L2 cache is 16-way set-associative. The figure shows the speedup and bandwidth increase when saving the tags of different number of MRU lines in each set. It indicates GHL-provides similar speedup compared to saving 4 or more tags with less bandwidth consumed. It is true that GHL-prefetching has advantages due to the feedback mechanism. However, a cache line is not an appropriate location for keeping prediction or feedback information since it could often be replaced by other cache accesses and is less likely to survive context switches. We also ex plored saving the tags of the blocks touched before a context switch but the resulting

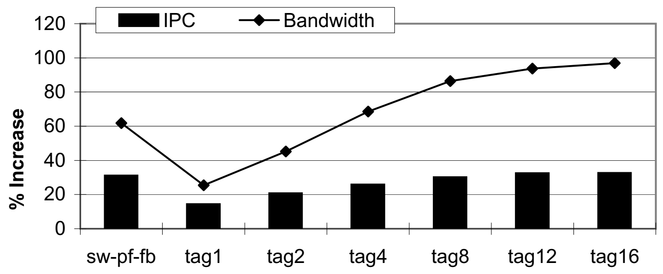


Fig. 9. Average speedup across all benchmarks for GHL-prefetching (*sw-pf-fb*) and saving tags (*tag**). The bars show the speedup while the line shows the percentage of memory bandwidth increase.

performance is significantly worse than GHL prefetching. This is expected since the GHL scheme retains information across several context switches.

VI. Conclusions

Among the various costs of a context switch, its impact on the performance of L2 caches is the most significant because of the resulting high miss penalty. In this paper, we propose mitigating this impact by prefetching into the L2 cache the data a program was using before it was swapped out. This is effectively restoring a program's locality destroyed by the context switch.

We use a Global History List to track which blocks are accessed when a program is running. When the OS decides to swap it out, these blocks are saved along with its context into memory. The next time the application runs, these blocks are loaded into a prefetching queue to guide prefetching. We carefully devise a placement policy to increase the lifetime of prefetches in the cache while minimizing the chance of replacing a line brought in by the process' demand accesses. To reduce the memory traffic incurred by prefetching, we design a feedback mechanism which eliminates useless prefetches by tracking the reuse patterns of prefetches, resulting in a significant memory bandwidth usage reduction.

Experimental results show that our scheme achieves 36% average speedup over no prefetching and 11%, and 24% average speedup in the presence of nextline and stride prefetchers respectively. In addition, the proposed feedback mechanism brings the extra memory traffic overhead down to 60% with a slightly lower speedup of 31% over no prefetching. GHL-prefetching with feedback not only outperforms traditional prefetching mechanisms, it does so while generating considerably lower extra memory traffic. These results show that context prefetching greatly reduces the impact of context switches on L2 cache performance, even outperforming a significantly larger L2 cache.

References

[1] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *24th Annual International Symposium on Computer Architecture*, June 1997.

[2] J. E. Smith and W.-C. Hsu, "Prefetching in supercomputer instruction caches," in *Proceedings of Supercomputing*, Nov. 1992.

[3] T. F. Chen and J. L. Baer, "Effective hardware-based data prefetching for high performance processors," *IEEE Transactions on Computers*, vol. 5, no. 44, pp. 609–623, May 1995.

[4] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 96.

[5] RedHat, IBM, Intel, and Hitachi, "http://sourceware.org/systemtap/." [Online]. Available: http://sourceware.org/systemtap/

[6] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," U. of Wisconsin, Madison, Technical Report CS-TR-97-1342, June 1997.

[7] G. Hamerly, E. Perelman, and B. Calder, "How to use simpoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, Mar. 2004.