PFetch: Software Prefetching Exploiting Temporal Predictability of Memory Access Streams [*]

Jaydeep Marathe and Frank Mueller

Computer Science, North Carolina State University, Raleigh, NC 27695, mueller@cs.ncsu.edu

ABSTRACT

CPU speeds have increased faster than the rate of improvement in memory access latencies in the recent past. As a result, with programs that suffer excessive cache misses, the CPU will increasingly be stalled waiting for the memory system to provide the requested memory line. Prefetching is a **latency hiding** technique that tackles this problem. If the address of the memory line that misses in cache can be predicted sufficiently in advance, it can be prefetched into the cache before it is accessed, reducing the effective latency of that access.

In this paper, we develop a novel software-only data prefetching scheme that works at the instruction level and exploits predictability in the access stream to prefetch memory lines accessed in the future. Working at the instruction level gives us a **global** view of memory access patterns across function, module and library boundaries. Conceptually, our scheme monitors the memory locations being accessed by loads and stores as well as their **contents**. It tries to find instances of **predictability** such that the address of a load miss can be pre-determined from a limited number of past accesses.

We make the following contributions in this work. First, we present a novel prefetching strategy that **unifies** and generalizes a number of past approaches that each target a specific source of address predictability. Specifically, our scheme unifies all these past approaches: next-line prefetching, self-stride prefetching, "intra-iteration" stride prefetching and same-object prefetching. In addition, it extends and generalizes the SPAID scheme for pointer and call-intensive programs. Second, we present a new threshold-based approach that addresses the issues of **prefetch accuracy**, **prefetch timeliness** and **prefetch redundancy**. Third, we assess our scheme both with a cache simulator and on a real machine where we evaluate it with hardware performance counters.

Overall, we demonstrate that a significant reduction in L1 cache misses can be achieved for several benchmarks on a real machine with our approach.

1. INTRODUCTION

In the recent past, processor speeds have been increasing at a much faster pace than improvements in memory access latencies. As a result, the cost of a cache miss in terms of processor cycles keeps increasing. Due to limited out-of-order window sizes of contemporary processors, a load miss in the second or further levels of cache will typically stall the processor as it runs out of parallel instructions to process. Consequently, the overall wallclock time for applications is often dominated by the efficiency of their memory access patterns.

Prefetching is a *latency hiding* strategy that attacks this problem. If the address of a load that misses in cache can be predicted sufficiently early, then the corresponding memory line can be prefetched into the cache well in advance of its access. If the prefetch completes before the load, then the erstwhile miss would be transformed into a hit.

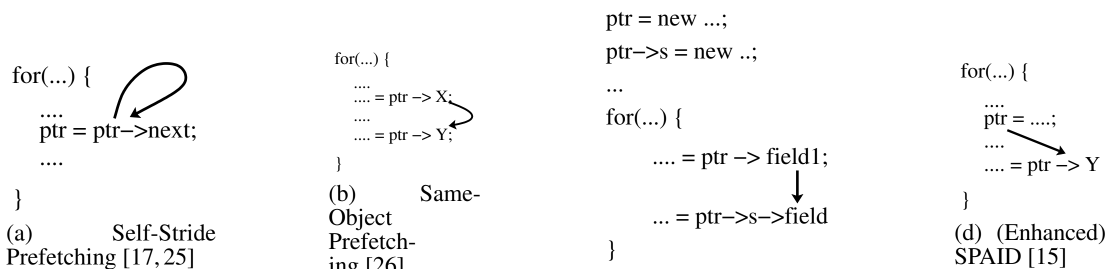
In this work, we focus on integer-intensive programs that typically contain few regular array accesses. Software prefetching algorithms for arrays accesses are well established (*e.g.*, [18]), and we do not target these in our approach[1].

Conceptually, our scheme monitors the memory locations accessed by loads and stores, as well as their *contents*. It tries to find instances of *predictability* such that the address of a load miss can be pre-determined from a limited number of past accesses. Our scheme is based on offline analysis using profile feedback. First, the program is run with a small training data set, and an annotated trace of memory accesses is extracted. This trace is analyzed offline for detecting predictability, and a set of *prefetch predictors* is generated. The prefetch predictors are used to place explicit software prefetch instructions directly in the assembly code of the program. In contrast to earlier hardware solutions [19,27], our scheme operates completely in software, and we present results from an implementation on a contemporary processor platform.

Our scheme has several advantages over past work. By examining the overall memory access streams of the executing program, we get a *global* view of a program's memory access pattern across function and module boundaries. This is hard to achieve for a static compiler, especially with irregular integer programs due to aliasing and input-dependent control flow. At the same time, our analysis is powerful and general enough to encompass and unify multiple separate approaches from past work. Specifically, our scheme unifies all these past approaches: next-line prefetching [13, 24], self-stride prefetching [8, 17, 20, 25], "intra-iteration" stride prefetching [11] and same-object prefetching [26]. In addition, it extends and generalizes the SPAID scheme [15] for pointer and call-intensive programs. A detailed discussion of related work is presented in Section 4. Examples of the access patterns targeted by our scheme are shown in Figure 1, along with the past work that addresses that pattern. The bold arrows depict the source and target of prefetching. Our scheme does not target each pattern *specifically*. Instead, all these patterns can be quite elegantly handled by a standard approach to analyzing memory accesses. Figure1(a) shows an example of self-striding that typically arises in pointer-chasing code where consecutive instances of the data structure tend to be placed at regular offsets from each other. Figure 1(b) shows "same-object" prefetching, where different fields of an object are frequently accessed close together in time. Since field layout is statically determined, the address of any field can be computed if the address of any other field of the same object is known. Figure 1(c) demonstrates "intra-iteration" stride prefetching. In many cases, data structures are allocated at the same time as their children. As a result, the address of the children fields can be predicted

[*]This work was supported in part by NSF grants CCR-0237570 (CAREER), CNS-0410203, CCF-0429653, an IBM faculty award and a grant by the Humboldt Foundation.

[1]Our baseline executable for performance comparison has compiler-inserted software prefetch instructions for array accesses.



(a) Self-Stride Prefetching [17, 25]

(b) Same-Object Prefetching [26]

(c) Intra-Iteration Stride Prefetching [11]

(d) (Enhanced) SPAID [15]

Figure 1: Example Patterns

from the parent object's address or the address of some other child. Inagaki *et. al* demonstrate that this occurs frequently with Java programs [11]. Finally, Figure 1(d) shows a generalized scenario targeted by a SPAID-like scheme. If the field dereference (ptr→y) is sufficiently distant from the pointer initialization, then the field can be prefetched in advance. Lipasti *et. al* only target pointers being passed as function arguments at call sites and also do not consider *offsets* from the pointer, as typically occurs with the dereferencing of a *field* of an object or a structure. We have generalized SPAID to consider *any* pointer load or store as a potential prefetch source. In addition, we consider load misses to a contiguous region of memory *around* the pointer as prefetch targets. This supports the common case of field dereferences that the original SPAID scheme did not consider.

We make the following contributions in this work:

- We present a novel software-only prefetch scheme that finds miss address predictability by monitoring the memory accesses of the program at the instruction level.
- Our approach unifies and encompasses several separate past efforts that each targeted different sources of predictability.
- We enhance and generalize the SPAID scheme that targets pointer dereference misses at function call sites.
- We also present novel measures to filter prefetch predictors based on *prefetch accuracy*, *prefetch timeliness* and *prefetch redundancy*. We are not aware of any past work that uses this approach.
- We implemented our scheme on a real machine and evaluate its performance with hardware performance counters.

2. FRAMEWORK

We are looking to exploit the *predictability* of memory access streams for reducing the number of load cache misses. We monitor the effective address (EA) generated by load and store instructions. In addition, we also monitor the *contents* of the memory location being accessed (EA_Contents) for loads/stores that might be accessing pointers.[2] Conceptually, for each load miss, we consult the recent history of memory accesses, constrained by a window size, to see if the load miss address was *predictable* using the captured EA, EA_Contents information of a retired dynamic instruction P. If so, then the load miss can be potentially transformed into a hit by inserted prefetch instructions that use the EA/EA_Contents value of P to prefetch data into the cache. In addition, we also try to address *prefetch timeliness* by checking whether the load miss is too *close* temporally to the predictor instruction P. If the load miss is found to be too close, the prefetch would not be useful.

For a target benchmark, we proceed through the following stages. First, we generate the assembly source code for the benchmark and use an annotation tool that we developed in-house to instrument memory access instructions. In addition, we insert instrumentation to maintain a pseudo "instruction counter" that conceptually increments after every instruction is issued[3]. We call this variable the *instruction distance (Inst_Dist)* counter. The idea is to tag each traced memory access with the Inst_Dist counter value. Later, during analysis, this helps in improving prefetch timeliness by detecting if prefetch predictors may be issued too close (temporally) with respect to the target load miss instruction, in which case the prediction would not be useful. Our experiments were performed on the Power architecture. We considered annotating the memory accesses with the values from the hardware high-precision timebase register in place of our software instruction distance counter. However, reading the timebase register is too expensive (≈1000 cycles for back-to-back reads of the timebase register), which significantly distorts the actual number of cycles between memory accesses. On other architectures (*e.g.*, Itanium2) the cost of reading the high precision timer is much lower. This may make it feasible to use the timebase register in place of the software instruction distance counter.

In the second step, we run the instrumented program and collect the memory access trace. Third, we analyze the trace using our framework and generate *prefetch predictors*. Each prefetch predictor is a tuple <IP, EA/EA_Contents, Delta>. 'IP' is the unique identifier for a load or store instruction. 'EA/EA_Contents' describes whether the effective memory address accessed by the instruction or the contents of that memory address should be used for prediction. Finally, 'Delta' is a constant value that denotes the offset from EA/EA_Contents to the memory line that needs to be prefetched. In the final phase, we insert a prefetch snippet just after the target instruction indicated by the prefetch predictor IP. The prefetch snippet issues a prefetch for the address EA/EA_Contents + Delta using the "Data Cache Block Touch" (`dcbt`) instruction. All these steps are completely automated.

Even with train data sets, the number of loads and stores in the full trace is very large. We therefore implemented *bursty tracing* to reduce the number of samples in the trace. Bursty tracing used a

[2]Our experiments were performed on the Power architecture where pointers are usually accessed using 32-bit `lwz` and `stw` instructions and their variants.

[3]We reduce the overhead by appropriately updating this counter only at basic block boundaries and before memory access instructions.

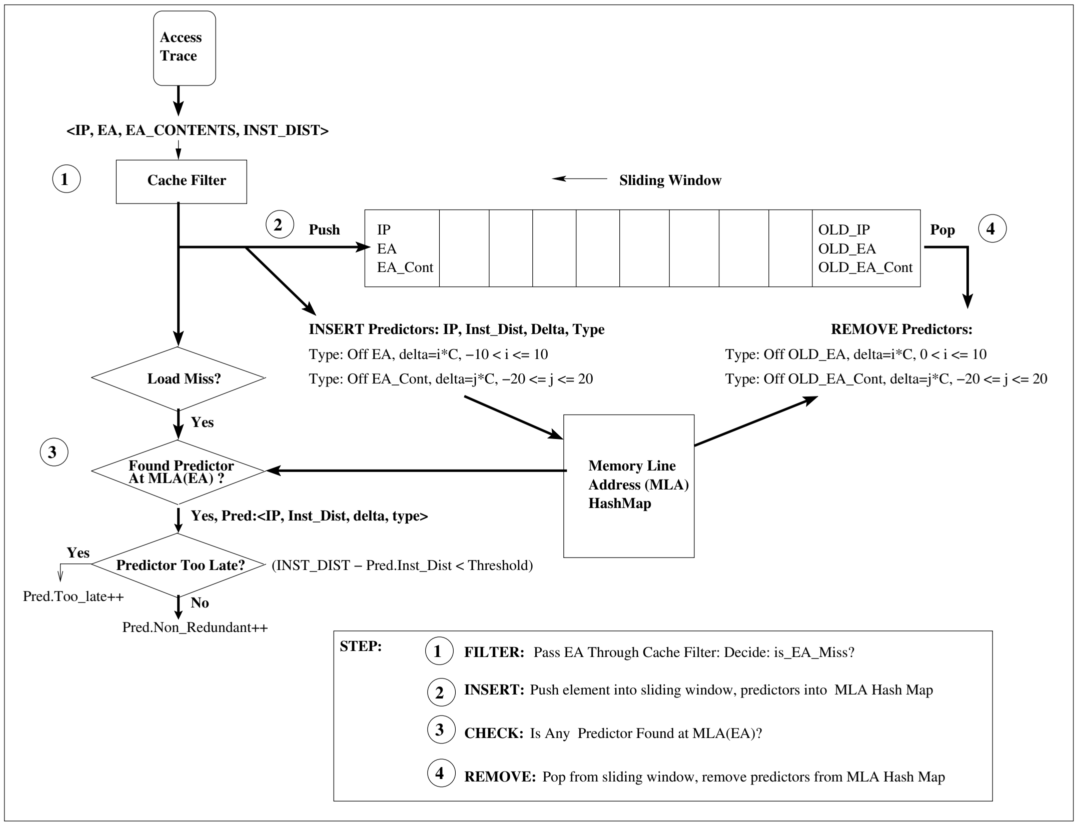

Access Trace
<IP, EA, EA_CONTENTS, INST_DIST>
1
Cache Filter
Sliding Window
2
Push
IP
EA
EA_Cont
OLD_IP
OLD_EA
OLD_EA_Cont
Pop
4
INSERT Predictors: IP, Inst_Dist, Delta, Type
Type: Off EA, delta=i*C, −10 < i <= 10
Type: Off EA_Cont, delta=j*C, −20 <= j <= 20
REMOVE Predictors:
Type: Off OLD_EA, delta=i*C, 0 < i <= 10
Type: Off OLD_EA_Cont, delta=j*C, −20 <= j <= 20
Load Miss?
Yes
3
Found Predictor At MLA(EA) ?
Memory Line Address (MLA) HashMap
Yes, Pred:<IP, Inst_Dist, delta, type>
Yes
Predictor Too Late?
(INST_DIST − Pred.Inst_Dist < Threshold)
Pred.Too_late++
No
Pred.Non_Redundant++
STEP:
1 FILTER: Pass EA Through Cache Filter: Decide: is_EA_Miss?
2 INSERT: Push element into sliding window, predictors into MLA Hash Map
3 CHECK: Is Any Predictor Found at MLA(EA)?
4 REMOVE: Pop from sliding window, remove predictors from MLA Hash Map

Figure 2: Overall Framework

duty cycle of 10%, each burst containing 2 million accesses [4].

We shall now describe our analysis in more detail. Figure 2 shows the steps in the analysis. We maintain a cache simulator "filter" that models the first-level L1D cache. The filter tags each memory access as a hit or a miss. Only load misses are targeted for prefetch prediction. All memory accesses are considered as potential prediction sources. For each memory access described by <IP, EA, EA_Contents, Inst_Dist>, we generate candidate predictors off both EA and EA_Contents. The idea is to keep track of a fixed contiguous region of memory *around* the EA and EA_Contents address for a certain number of accesses in the future. If any of these future accesses are load misses in this region, then those load misses can be prefetched using the value of EA or EA_Contents and a constant offset value *Delta*, as described before. The size of the contiguous region is configurable. We empirically determined that region sizes of 10 and 20 cache lines in each direction (positive/negative) gave good results for EA and EA_Contents, respectively[5]. A smaller region potentially reduces the number of load misses detected as suitable for prefetching, but with the benefit of reduced analysis overhead.

How long should we keep the predictors around? This is a configurable parameter and is implemented by a *sliding window* scheme. The length of the sliding window decides the number of future accesses during which the predictors remain active. The trace record (IP, EA, EA_Contents) is put into the sliding window when the candidate predictions are generated. When the trace record pops out of the sliding window, the predictors are removed. The candidate predictions are hashed by their memory line addresses and stored in the *Memory Line Address Hash Map (MLA map)*. For each load miss, the load miss address is checked in the MLA map. If there are existing predictions for the load miss address, each predictor is considered further for validity. The Inst_Dist values of the predictor and the load miss are checked. If the difference in instruction distance is below a certain threshold, the prediction is classified as "too late". Otherwise the prediction is considered useful (Non_Redundant).

After the entire trace has been processed, we prune the predictors using various thresholds. First, we consider the *accuracy* of predictions. Predictors are considered highly accurate if the predicted cache line is either already in the cache or is accessed within the sliding window before the predictor was removed (when popping off accesses from the end of the sliding window). This is known as the *seen_ratio*.

seen_ratio = (# predicted line seen in sliding window or already in cache) / (# predictions)

The idea is to reduce the overhead of *useless prefetches* by only selecting predictors with high accuracy. Useless prefetches are very expensive because they cache data that is seldom accessed, they may evict frequently accessed memory lines from the cache, and they incur overhead for executing the prefetch snippet.

Our second threshold addresses the issue of *prefetch timeliness*. The Inst_Dist difference between a predictor and its target load miss denotes the number of dynamic instructions issued between them. This is a conservative lower bound on the number of *processor cycles* between these two events since it does not account for multi-cycle instructions, such as loads that miss in cache. If this differ-

[4]In other words, we captured 2 million accesses and then ignored the next 18 million accesses.

[5]In Figure 2, `C` denotes the cache line size.

Table 1: Benchmarks and data sets

Benchmark	Suite	Train Data Set Arguments	Ref Data Set Arguments
181.mcf	SPEC CPU2000	train/inp.in	ref/inp.in
300.twolf	SPEC CPU2000	train	ref
255.vortex	SPEC CPU2000	bendian.raw	bendian1.raw
175.vpr	SPEC CPU2000	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
197.parser	SPEC CPU2000	2.1.dict -batch < train.in	2.1.dict -batch < ref.in
boxedsim	-	-n 500 -t 0.75	-n 1000 -t 1.0
ft	PtrDist	10000 40000	10000 40000
bh	Olden	8192 1	65536 1
bisort	Olden	10000000 1	20000000 1
em3d	Olden	10000 100 75 1	1000000 100 75 1
health	Olden	5 3000 1	5 4000 1
mst	Olden	4096 1	8192 1
treeadd	Olden	24 1	26 1
tsp	Olden	2000000 1	3000000 1
voronoi	Olden	500000 1	10000000 1

ence is lower than a certain threshold, we consider the predictor to be "too close" temporally to the target load instruction that missed. Hence, the prediction is not useful. The *too_late_ratio* is calculated as:
too_late_ratio = (# predictions classified as too late/ # predictions)
If the too_late_ratio is above a certain value, the predictor is pruned.

Finally, we attempt to reduce the number of *redundant* prefetches. Consider, for example, a structure S with three elements, A, B and C, that reside in different cache lines with the fields always accessed in the order S.A, S.B and S.C. If S.C is a miss, then the load miss address can be predicted using both S.A and S.B. However, the second predictor (off S.B's EA) is redundant and should be pruned. The redundant predictors are pruned as follows. First, the set of predictors pruned using the other thresholds discussed above is generated. Then, the trace is re-processed from the start and passed through a new cache filter. At the end, the redundancy_ratio for each predictor is calculated as follows:
redundancy_ratio = (# predicted line already in cache / # predictions)
If the redundancy_ratio is above a certain threshold, then the predictor is pruned.

At the end of the analysis, we have obtained a set of predictors that are highly accurate, have good timeliness and are highly relevant. These predictors are inserted back into the assembly source code as described earlier. Our current scheme is based on annotating assembly code, though our framework can be implemented in the back-end of a compiler as well. The prefetch snippets as well as the instrumentation snippets need at least two free registers. In our current implementation, we reserve two registers using a compiler flag[6] when generating the target program's assembly source. This requirement can be removed by selecting dead registers using live variable analysis [17].

Our scheme is not targeting load misses incurred by regular array accesses. Prefetching algorithms for such accesses are well established. Our compiler (gcc) is able to generate prefetches for such array accesses. Our analysis takes into account the effect of this compiler-generated prefetching in the following way. When tracing memory accesses, we also trace the prefetch instructions inserted by the compiler. During analysis, the effect of these prefetch instructions is simulated by the cache filter. It is important to do this because, otherwise, our analysis might generate prefetch predictors for array accesses that are also targeted by the compiler-generated prefetch instruction. Our predictors would be redundant in this case.

3. EXPERIMENTS

We evaluated our framework for a set of memory-intensive benchmarks shown in Table 1. The benchmarks have been selected from the SPEC CPU 2000 [9], Ptrdist [1] and Olden [3] suites, except for boxedsim [4]. We selected benchmarks that had significant L1D cache miss rates. Smaller data sets were used for the training phase, while larger data sets were used for measuring execution benefits. For the boxedsim benchmark and the benchmarks from the Olden and PtrDist suites, we increased the data set sizes from their default values to make the programs run sufficiently long.

3.1 Procedure

All experiments were conducted on a multi-user p655 Power4 SMP system. The system has 8 processors but we used only one of them since our programs are single-threaded. Each Power4 processor has a 16 KB 2-way associative L1D cache, 1.5 MB 8-way associative unified L2 cache and a 32 MB 8-way associative L3 cache. The L1D and L2 cache lines are 128 bytes wide.

For each program, assembly code was generated by the compiler. We used the gcc compiler with high optimization settings (-O3 -mpower -fprefetch-loop-arrays[7]). In addition, as stated before, we also reserved two registers for our prefetch snippet in the generated assembly code using the flags "-ffixed-r15 -ffixed-r16". The executable built from this assembly source is our baseline for performance evaluation. In the profiling/analysis pass, this assembly source was instrumented for bursty tracing, and the generated traces were analyzed using our framework. The generated predictors were inserted into the assembly source that was assembled and linked to create our modified executable for evaluation.

We only had access to a shared multi-user Power4 platform. The shared usage caused variability in our performance measurements. In order to address this problem, each measurement run was re-

[6]We used gcc for our experiments. The corresponding flags for gcc are -ffixed-r15 -ffixed-r16

[7]The "-fprefetch-loop-arrays" directs the compiler to insert explicit prefetch (`dcbt`) instructions for loop-resident array accesses.

peated for 10 times, and the *minimum* values were chosen for comparison.

3.2 Self-Striding Comparison

Past related work on data prefetching has focused on *self-striding* [17, 25]. We implemented a self-striding scheme and compared its performance to our framework. The implementation of the self-striding scheme is based on Wu *et al.'s* scheme [25]. For each load instruction, the load access trace generated is examined. Up to 4 most frequent strides are selected. Strides that account for less than 25% of the accesses at that instruction are pruned. We keep track of the average instruction distance B (Section 2) between consecutive accesses at an access point. If the access point is in a loop, this represents an approximate estimate of the number of cycles that the loop body takes to execute. If the cache miss penalty is L cycles, the prefetch multiplier K is calculated as:
K = min($\lceil L/B \rceil$, C) , where C is the maximum distance multiplier. We use L=120, C=10. 120 cycles is the latency for an L2 miss on the Power4. The value for C is chosen to be similar to past work reported in Wu *et al.* [25] to permit a fair comparison.

The prefetch predictor has a distance of K*stride and is inserted into the assembly source code as described before.

3.3 Measurement Metrics

We use two metrics — L1D load cache misses and the total number of processor cycles executed by the program. We used hardware performance counters to measure these two metrics with the `hpmcount` performance monitoring tool (events PM_LD_MISS_L1 and PM_CYC, respectively). The performance measurements were undertaken with a different and larger data set compared to the training data set used for tracing and analysis (Table 1).

In addition, we also measured the L1D load cache misses on a cache simulator using only the train data set for both the original and predictor-inserted variants. The simulator is event-based (*i.e.*, it does not model timing), and the cache size is same as the as the real machine's L1D cache (32 KB). It also does not model the hardware stream prefetcher available in the Power4 platform. In spite of its limitations, the simulator output provides a useful indication of the magnitude of *potential* miss rate savings possible, as will be shown.

3.4 Analysis

Table 2 shows the values of the parameters used during our analysis. Thus, predictors were pruned if they satisfied any of the following conditions: the predicted memory line was "seen" less than 85% of the time, the prediction was too late more than 60% of the time or the prediction was redundant (*i.e.*, the target memory line was already in cache) more than 95% of the time. The sliding window size was 450, and candidate predictors were generated for up to 10 and 20 cache lines in each direction for EA and EA_Contents, respectively.

Table 2: Analysis Parameters

Parameter	Value
seen_threshold	0.85
too_late_threshold	0.6
redundancy_threshold	0.95
inst_dist_threshold	64
sliding_window_size	450
ea_locality_lines	10
ea_contents_locality_lines	20

Figure 3 shows the percentage reduction in L1D cache misses as reported by the simulator processing the traces from the train data set. It gives a useful indication of the benchmarks where our technique is applicable. Figures 4 and 5 show the percentage reduction in (a) L1D cache misses and (b) processor cycles, respectively, on the real machine, as reported by the hardware performance counters with the reference data set.

Consider the simulation results shown in Figure 3. In 8 benchmarks (out of 14), our scheme is able to reduce the L1D cache miss rates by 5% or more with more than 20% reduction in 5 benchmarks (mcf, parser, em3d, mst, treeadd, tsp). Self-striding performs significantly worse in 6 of these benchmarks (vortex, parser, boxedsim, em3d, mst, tsp), shows comparable performance in 2 (mcf, treeadd) and is significantly better for FT. FT is discussed in more detail below. For the other benchmarks, the difference in performance with respect to self-striding can be attributed to the fact that our scheme targets multiple different sources of predictability (Section 1) while self-striding only focuses on the regularity of the access stream at individual access points.

Consider the actual reduction in L1D misses obtained with the reference data sets in Figure 4. We observe that there are significant savings for many benchmarks, but the values are uniformly lower as compared to the simulator predictions (except for vortex, where they are higher). For our scheme, vortex and tsp show a significant reduction ($>$ 30%) while mcf, boxedsim, em3d and mst exhibit appreciable reductions ranging from 4.9% to 17%. In comparison, self-striding has significantly worse performance in 4 of these benchmarks (vortex,em3d, mst, tsp), comparable performance in 2 (mcf, boxedsim) and performs significantly better for FT. Some benchmarks that show significant improvement with the simulator do not show a corresponding improvement on the real machine with the larger reference data set (parser, mst, treeadd). The potential reasons for this discrepancy are discussed below.

Figure 5 shows the corresponding reduction in processor cycles. The performance results are mixed. 6 benchmarks (mcf, vortex, vpr, bisort, em3d, tsp) show appreciable reductions ranging from 2% to 7.6%. However, 4 benchmarks (parser, boxedsim, ft, treeadd) experience slowdowns ranging from -2% to -12%. Except for FT, self-striding always performs worse for all benchmarks, *i.e.*,no benchmark shows an improvement of 2% or more. For self-striding, mcf achieves significant L1D miss reductions but almost no processor cycle reductions. We suspect that the cause is suboptimal instruction scheduling for the prefetch snippet, as discussed below.

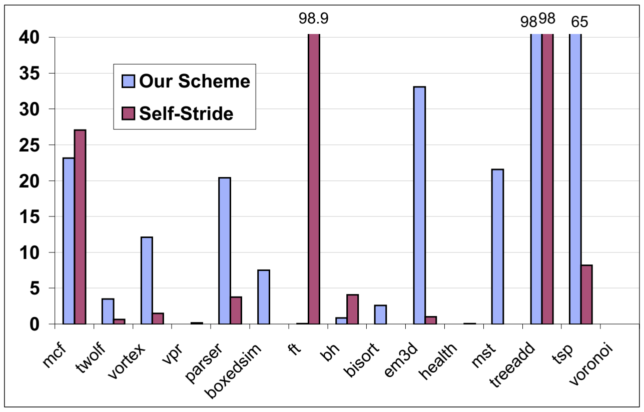


Figure 3: Simulator: % Reduction in L1D load misses (Training data set)

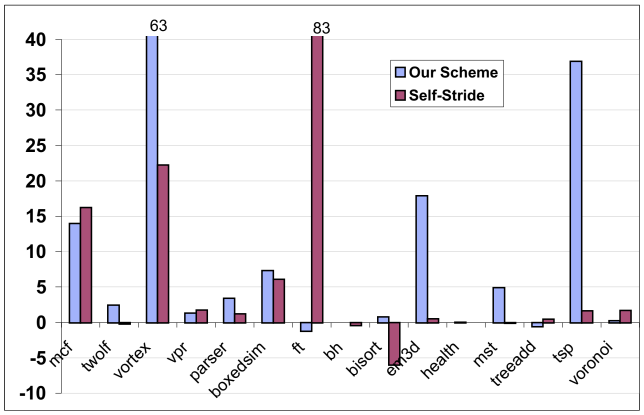


Figure 4: H/W: % Reduction in L1D load misses (Reference data set)

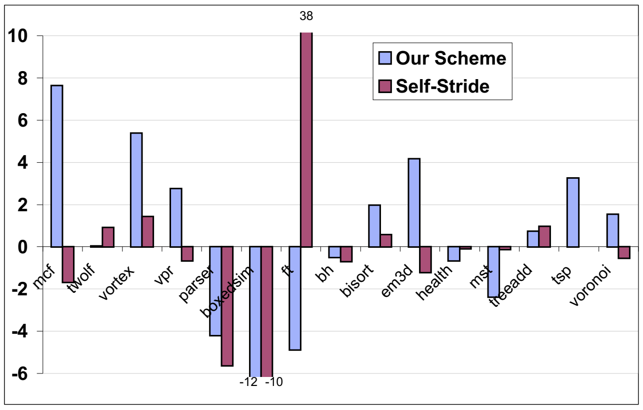


Figure 5: H/W: % Reduction in processor cycles (Reference data set)

We have seen that for many benchmarks, the simulator results are much better than the results on the actual machine. In addition, for several benchmarks the L1D miss reductions on the real machine do not correspond to a reduction in processor cycles. In fact, some benchmarks show a degradation in performance. What could account for these anomalies? To attain a definitive answer, we would need a cycle-accurate simulator that models our target (commercial) Power4 processor. There are several plausible explanations:

- **Simulator**: Our simulation results are expected to be *optimistic*, since the simulator is event-based and does not model prefetch timeliness. In addition, it does not currently model the hardware stream prefetcher present in the Power4 platform. Some of the predictable prefetches may also be recognized by the hardware prefetcher, reducing the observed L1D miss savings on the real machine.

- **Prefetch snippet Cost**: We do not account for the cost of the prefetch snippets. This cost can overwhelm the benefits of reduced misses if there are many redundant prefetches.

- **Prefetch scheduling**: Our current approach is based on annotating the assembly source code. In the course of our experiments, we found that scheduling of the prefetch snippets has a big impact on the overall processor cycles. Implementing our scheme in the backend of the compiler would solve this problem.

3.5 FT

FT illustrates a potential weakness of our scheme. Self-striding performs exceptionally well for FT while our scheme did not find significant prefetch predictors. In this benchmark, there is a single load access in a tight pointer-chasing loop that accounts for the bulk of the load misses. The access point exhibits a regular stride of -120. Self-striding is able to detect this stride and generates a prefetch predictor distance of -1200 (since the maximum distance multiplier, C, is 10) off the effective address (EA) of the access point. Recall that our sliding window analysis only generates predictors for a *limited* region around the access point's EA. In our experiments, this was set to 10 cache lines, *i.e.*, the predictors generated had prefetch distances of -1152, -1024,...-128. However, all these prefetch predictors were *classified as "too late" because of the tight nature of the pointer-chasing loop nest. Hence, they were rejected.*

Thus, in this case, self-striding was able to generate prefetch predictors while the sliding window analysis failed to do so. The cause was that the high instruction distance threshold (64) was *too pessimistic* — it did not consider the number of cycles the processor was stalled due to a cache miss but only considered the actual number of instructions issued between the candidate predictor and the target dynamic load instruction. In further experiments, lowered the instruction-distance threshold below 64. This helped FT gain performance but adversely effected other benchmarks. It appears that a more adaptive scheme to adjust the instruction-distance threshold would be needed to be universally affective across different codes.

4. RELATED WORK

Sair *et al.* performed a simulation-based study to classify program load misses into *next-line*, *stride*, *same-object* and *pointer* misses [23]. A significant fraction of the load misses was found to be of one of the first three types for many of the SPEC benchmarks. It is precisely these misses that are targeted by our framework. Hardware implementations of next-line and stream prefetching have been proposed earlier [13,24]. The Power4 platform has a stream-based hardware prefetcher. However, this prefetcher needs a "warm-up" period of a certain number of misses to consecutive cache lines before it starts to prefetch the stream. Hur *et al.* propose a memory-side prefetcher for the Power5+ system that targets such short-length streams [10]. In contrast, our approach is completely software-based and uses explicit prefetch instructions to target such streams successfully.

Stride-based prefetching has been explored both in hardware and software. Unlike our scheme, much of the past work has focused on the striding regularity of *individual* access points, *i.e.*, they do not consider predictability among accesses from different load/store instructions. Wu *et al.* [25] and Luk *et al.* [17] describe such software self-striding schemes. Hardware stride prefetchers have also been proposed [8, 20] and implemented in the Pentium microprocessor. As shown in this work, our scheme performs significantly better than self-stride prefetching in isolation for most benchmarks. This is because our approach is able to detect multiple other sources of predictability in addition to most cases of self-striding. However, in a few situations, self-striding may perform better than our scheme (*e.g.*, for FT). Zhang and Torrellas propose a hardware scheme to group fields or objects that are used together, and prefetch all these at the same time [26]. In comparison, our scheme will also recognize groups of field accesses that occur together and target them for prefetching if they miss frequently in cache. However, our scheme is implemented completely in software.

Lipasti *et al.* propose SPAID, a prefetching heuristic for pointer and call intensive programs [15]. SPAID inserts prefetches for the target of pointer parameters at call sites on the assumption that the pointed object will be dereferenced soon. Our scheme extends SPAID in that *any* pointer load is analyzed as a potential prefetch candidate, not just the pointers at call sites. In addition, when looking for predictable cache misses, we look at a large contiguous area of memory *around* the candidate pointer while SPAID will only prefetch at most one cache line (the one pointed to by the pointer). Our experiments indicate that our approach is better because pointers pointing at objects or structures are often used to deference *fields* of an object that span multiple cache lines. Finally, our detailed trace information allows much better pruning of harmful or useless prefetches. The dynamic instruction distance allows us to model *prefetch timeliness* for SPAID-like prefetch predictors, pruning away those that would be too late to be useful. The redundancy threshold prunes away predictors whose targets are most often already in cache.

Inagaki *et al.* present a software prefetching approach for targeting both “inter-iteration” (self-striding) and “intra-iteration” predictability (predictability across instruction points) [11]. This is the closest related work. During just-in-time compilation of a hot Java loop, the load dependence graph is constructed. An interpreter runs the first few iterations of the loop and attempts to determine stride patterns among the different loads in the dependence graph. This information is used to insert software prefetches, and speedups of up to 20% are reported on some benchmarks. Our scheme shares the goal of determining address predictability among multiple memory access points. However, our offline approach allows us to conduct a much more detailed analysis by accounting for prefetch timeliness and prefetch redundancy. Our scheme targets misses from *all* loads, not just the loop-resident ones. Finally, our scheme can potentially *group* multiple prefetches because it considers *memory lines* and not just the strides between two loads in a pair — since multiple fields typically share the same cache line.

Beyler and Clauss developed a binary rewriting framework to first measure latencies of loads using the time counter register of the IA64 before inserting prefetches where beneficial [2]. They periodically monitor program behavior to add and remove prefetches in response to their effectiveness and to changes of program behavior (phasing). After compensating for instrumentation overhead, they report 2-143% performance improvement. Results are hard to compare because (a) they lack prefetching under gcc while we enabled it and (b) since IA64 performance is known to more critically depend on memory performance than that of the Power architecture with hardware prefetching. In contrast to their work, ours determines predictors before inserting prefetches statically, which does not incur any runtime overhead.

Hardware schemes have been proposed that assess predictability in misses generated from different access points [19, 27]. In contrast, our approach is completely in software and is therefore more portable.

Finally, there are other prefetch schemes that are complementary to our approach. Software prefetching for regular array accesses in loop intensive programs is well established [18]. We do not target these misses but focus on other sources of address predictability in irregular integer-intensive programs. Prefetching of pointer chains, for example using Markov predictors, has been previously proposed in both hardware [6, 7, 12, 21] and software [5, 22]. Our scheme does not address this problem. Finally, helper-thread based approaches leverage additional hardware contexts to prefetch data for the main program [14, 16]. In contrast, we focus on prefetch instructions inserted *inline* in the target program that can be single-threaded or multi-threaded.

5. CONCLUSIONS AND FUTURE WORK

In this work, we have presented a novel software-only prefetch scheme that exploits predictability in the memory access stream to create prefetch predictors. Our approach unifies and extends several past approaches that each targeted a different source of address predictability. Our approach is fully automated, portable and uses novel threshold-based mechanisms for addressing prefetch accuracy, prefetch timeliness and prefetch redundancy.

We have evaluated our scheme on a real machine as well as a cache simulator. We have shown that significant reductions in L1D load cache misses are achievable on real hardware with our approach. However, the performance improvements in terms of processor cycles has been lower than anticipated. Based on experience with microbenchmarks, we conjecture that *prefetch scheduling* has a significant impact on the overall processor cycles. In future work, we intend to port our framework to the back-end of a compiler, which will allow us to schedule prefetch snippets. In addition, we will port our framework to other memory constrained architectures, such as the Itanium2 platform.

6. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[2] J. Beyler and P. Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In *International Conference on Supercomputing*, pages 202–209, 2007.

[3] M. C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Princeton University, Princeton, NJ, USA, 1996.

[4] S. Chenney. Controllable and scalable simulation for animation, 2000.

[5] T. Chilimbi. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 191–209, June 2002.

[6] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[7] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 279–290, New York, NY, USA, 2002. ACM Press.

[8] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 133–143, New York, NY, USA, 1997. ACM Press.

[9] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[10] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, Washington, DC, USA, 2006. IEEE Computer Society.

[11] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 269–277, New York, NY, USA, 2003. ACM Press.

[12] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[13] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[14] D. Kim, S. S. wei Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 27, Washington, DC, USA, 2004. IEEE Computer Society.

[15] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 231–236, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[16] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[17] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link optimizer for the intel&#174;itanium&#174;architecture. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

[18] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Notices*, 27(9):62–73, 1992.

[19] 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*, page 96, Washington, DC, USA, 2004. IEEE Computer Society.

[20] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[21] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126, New York, NY, USA, 1998. ACM Press.

[22] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 111–121, Washington, DC, USA, 1999. IEEE Computer Society.

[23] S. Sair, T. Sherwood, and B. Calder. Quantifying load stream behavior. In *HPCA*, pages 197–, 2002.

[24] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[25] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 210–221, New York, NY, USA, 2002. ACM Press.

[26] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 188–199, New York, NY, USA, 1995. ACM Press.

[27] H. Zhou and T. M. Conte. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Trans. Comput.*, 54(7):897–912, 2005.