2014 43rd International Conference on Parallel Processing

A Case for Resource Efficient Prefetching in Multicores

Muneeb Khan, Andreas Sandberg and Erik Hagersten
Department of Information Technology, Uppsala University
Email: {muneeb.khan, andreas.sandberg, eh}@it.uu.se

***Abstract*—Modern processors typically employ sophisticated prefetching techniques for hiding memory latency. Hardware prefetching has proven very effective and can speed up some SPEC CPU 2006 benchmarks by more than 40% when running in isolation. However, this speedup often comes at the cost of prefetching a significant volume of useless data (sometimes more than twice the data required) which wastes shared last level cache space and off-chip bandwidth.**

This paper explores how an accurate resource-efficient prefetching scheme can benefit performance by conserving shared resources in multicores. We present a framework that uses low-overhead runtime sampling and fast cache modeling to accurately identify memory instructions that frequently miss in the cache. We then use this information to automatically insert software prefetches in the application. Our prefetching scheme has good accuracy and employs cache bypassing whenever possible. These properties help reduce off-chip bandwidth consumption and last-level cache pollution. While single-thread performance remains comparable to hardware prefetching, the full advantage of the scheme is realized when several cores are used and demand for shared resources grows.

We evaluate our method on two modern commodity multicores. Across 180 mixed workloads that fully utilize a multicore, the proposed software prefetching mechanism achieves up to 24% better throughput than hardware prefetching, and performs 10% better on average.

I. Introduction

As the gap between DRAM and processor speeds remains considerably large, memory latency remains a major bottleneck for system performance. The compute power of multicores is expected to keep growing faster than the off-chip bandwidth in future [16], which means that techniques used to hide memory latency should not waste memory bandwidth and shared cache space in order to be effective. Modern processors typically employ aggressive hardware prefetching to hide memory latency. Such prefetchers can sometimes improve single-threaded performance by more than 40% on modern commodity multicores. However, aggressive hardware prefetching often increases off-chip traffic significantly, sometimes more than twice the amount needed (Figure 5b). Such speculative prefetching helps maximize single thread performance, but can severely strain shared resources by i) polluting the shared last-level cache (SLLC), and ii) increasing off-chip bandwidth usage. Modern processors throttle down prefetching to avoid shared-resource wastage during contention. However our experiments show significant useless off-chip traffic even during full system utilization (Figure 7d).

Shared-resource conservation: Wasting shared resources directly impacts the performance of applications co-executing on neighboring cores. This suggests that prefetching approaches for multicores need to be especially focused on being accurate and resource-conserving. In this work we present such a resource-conserving software-only method aimed at improving multicore throughput performance in shared-resource constrained situations by i) improving single-thread performance with accurate prefetching, and ii) minimizing off-chip traffic and off-chip bandwidth consumption.We compare our method with state-of-the-art hardware prefetching in modern commodity multicores and show that multicore performance in highly-utilized systems improves significantly (up to 24%) when shared resources are conserved.

Low-Overhead mechanism: Prior work [1], [10], [15], [21] has investigated the use of software prefetching for improving single thread performance, which does not take in to account shared resources of multicore systems. In addition, most previous profile guided optimization (PGO) methods incur large profiling overheads. To make our approach more applicable, we propose the use of low-overhead runtime sampling and fast cache modeling. This enables us to identify memory instructions that frequently miss in the cache (referred as *delinquent loads*), and insert software prefetches for such delinquent loads that exhibit regular strides. Our approach is accurate and consumes significantly less off-chip bandwidth than hardware prefetching. The proposed technique is designed to work at the binary level, and could enable runtime optimization methods such as dynamic binary rewriting, so that optimizations are applicable even when the source is unavailable.

We make the following main contributions:

- We demonstrate a low-overhead method to identify delinquent loads, and do accurate prefetching for them in software. Our method uses low-overhead runtime sampling of a target application to gather runtime information. The runtime information is used as input to fast statistical cache modeling to accurately identify delinquent loads that can benefit from prefetching.
- We combine a cache-bypassing technique with our software prefetching method to lower off-chip traffic and cache useful data longer.
- We evaluate our method and compare with hardware prefetching on two high-performance processors. Our resource-efficient method achieves single-thread performance comparable to hardware prefetching, while reducing off-chip traffic by 44% on average. In a fully utilized multicore our method consistently achieves significantly higher throughput than hardware prefetching.

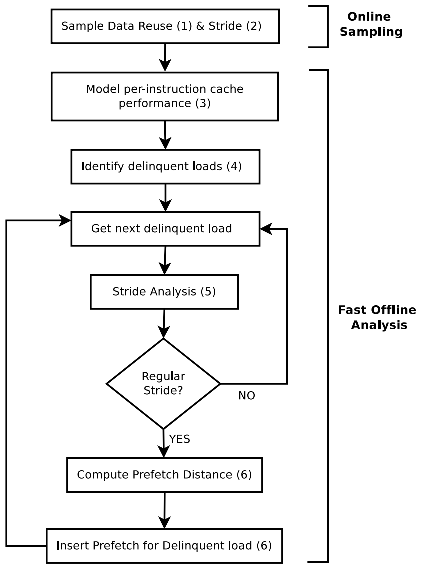

Sample Data Reuse (1) & Stride (2)
Online Sampling
Model per-instruction cache performance (3)
Identify delinquent loads (4)
Get next delinquent load
Stride Analysis (5)
Regular Stride?
NO
YES
Compute Prefetch Distance (6)
Insert Prefetch for Delinquent load (6)
Fast Offline Analysis

Figure 1: Optimization Framework Overview

II. Overview

We start by describing our framework that enables an effective resource-efficient prefetching method. The framework goes through several passes as shown in Figure 1. This section briefly describes these different passes. We discuss each step in detail in later sections. The first two steps are part of a single integrated sampling pass. The rest are steps in a *fast* offline analysis pass.

1) **Data reuse sampling:** Memory instructions are sampled randomly and sparsely. Data cache blocks accessed by sampled instructions are monitored for reuse. A reuse sample is recorded whenever a monitored cache block is re-accessed.
2) **Stride sampling:** The sampled memory instructions are monitored. Whenever re-executed, a stride sample is recorded as the difference of the current and previous memory address accessed by this instruction.
3) **Fast Cache modeling:** Data reuse samples recorded over the entire execution *(step 1)* are fed to a statistical cache model [4] which models per-instruction cache performance for given cache sizes. Figure 3 shows the modeled cache behavior for a sampled instruction in *mcf*.
4) **Delinquent load identification:** A *cost-benefit* analysis uses the modeled per-instruction cache performance *(step 3)* to determine if a load can benefit from software prefetching. Such loads are analyzed further.
5) **Stride analysis:** Stride samples *(step 2)* for each delinquent load *(step 4)* are analyzed to identify regular stride patterns. If a dominant stride pattern is observed, the identified delinquent load becomes a candidate for prefetching.
6) **Prefetching analysis:** Appropriate prefetch distance is computed and a prefetch instruction is scheduled for the delinquent load.

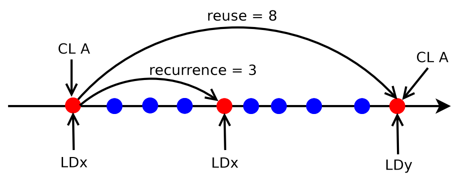


Figure 2: Data Reuse and Stride sampling - *Recurrence* is the number of memory references between successive executions of load LD$_x$. *Reuse distance* is the number of memory references between two accesses to sampled cache line (A).

III. Data Reuse & Stride Sampling

Our method starts by sampling the application at random execution points to record: 1) data reuse samples (repeated access to same cache line), and 2) stride pattern of sampled memory instructions. Figure 2 illustrates the sampling effect at a sampled execution point. To keep the sampling overhead small, the application is sampled very sparsely - with only 1 in every 100 000 memory references sampled.

Data reuse samples are used as input to the StatStack fast cache model [4] to model cache behavior for different cache sizes. Data reuse sampling captures *reuse distance* samples across the entire execution. *Reuse distance* is the number of memory accesses (not necessarily to different cache lines) between two consecutive accesses to the same cache line. Low overhead data reuse sampling frameworks developed by Sembrant et al.[19] show that data reuse samples can be captured over the entire execution of an application with an overhead of less than 20% over native execution. Their technique can briefly be described as follows:

1) The next memory reference to sample is randomly selected. Hardware counters are programmed to overflow and halt the execution at that instruction.
2) A watchpoint is set for the data cache line accessed by the instruction, after which the execution is resumed.
3) A trap occurs when the monitored cache line is accessed again. Hardware counters are used to measure the number of intervening memory references between the two accesses to the cache line (i.e. the reuse distance).

We extended the sampling framework to sample per-instruction stride and *recurrence* information (Figure 2), which is recorded by monitoring sampled instructions for re-execution. When execution stops for sampling, a breakpoint is set for the sampled instruction, and the execution is continued. When the instruction is executed again, the breakpoint fires and stops the execution. At this point the breakpoint is removed and 1) a stride sample is recorded as the difference of current and previously accessed data addresses, and 2) *Recurrence* is recorded as the number of intervening memory references between successive executions of the sampled instruction. The execution is then continued. Figure 2 illustrates the collective data reuse and stride sampling for a randomly selected load in the instruction stream. This additional functionality of monitoring strides can be added to the existing reuse samplers keeping the overall overhead below 30%.

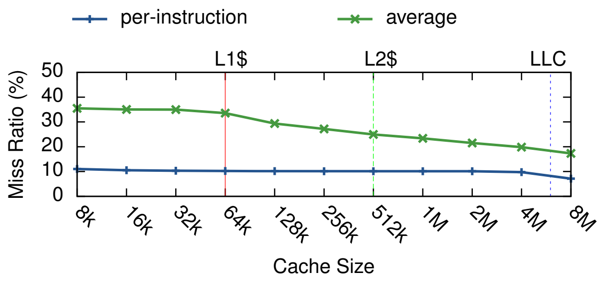


Figure 3: Miss Ratio Modeling - average miss ratio of *mcf* and miss ratio of a frequently executed *load* - both modeled by StatStack. The *L1*, *L2* and *LLC* cache sizes for AMD Phenom II are marked.

IV. Fast Cache Modeling

StatStack is a fast cache model that estimates miss ratios - *The fraction of memory accesses that miss in the cache of any size*. StatStack uses the sampled *reuse distribution* as input. Using a statistical approach it converts the reuse distance distribution to an estimated stack distance[1] distribution. The estimated stack distance distribution is then used to accurately model the overall application and per-instruction miss ratios. Figure 3 shows the average miss ratios for the entire run of *mcf* benchmark and a single load instruction in the benchmark, both modeled using StatStack[2]. Per-instruction miss ratio curves (Figure 3) are used by our analysis to determine the (delinquent) loads that frequently miss in the cache and can benefit from prefetching.

Since our method relies on statistical cache modeling to identify delinquent loads, it is extremely important that the model gives good coverage of memory instructions that frequently miss in the cache. To evaluate the coverage of StatStack (with 1 in 100,000 sampling rate), we compared it to a Pin based [8] functional cache simulator. The simulator provided a baseline miss ratio for every memory instruction. The data cache was configured as 64 kB, 2-way associative with cache line size of 64 B (AMD Phenom II configuration). When comparing with simulation, StatStack identified 88% of all misses on average across the SPEC CPU 2006 benchmarks. We also evaluated the model against a 512 kB L2 cache and found 94% misses covered on average. We can safely say that the most important delinquent loads are identified by the cache model. StatStack is extremely fast compared to simulations, and typically takes less than a minute to model average and per-instruction miss ratios for several cache sizes.

V. Delinquent Load Identification

This section describes a method for identifying delinquent loads that benefit from prefetching. The cache model provides per-instruction miss ratios for the specified L1, L2 and LLC sizes, as shown in Figure 3. We use the miss-ratios to quantitatively determine if inserting software prefetch for a load will improve performance or not. For example, if a load hits in the L1 cache 90% of the time and 10% of the time in L2. A software prefetch for this instruction will be usefully executed 1 in 10 times to remove the L1 misses. Assume that it takes 1 cycle to execute a software prefetch instruction and the latency to the L2 cache is 5 cycles. Then we will end up executing 10 prefetch instructions costing us 10 cycles and in return saving only 5 cycles (saving a single L1 miss). Such software prefetching becomes an overhead and hurts performance instead of improving it. We employ a cost-benefit analysis that helps us avoid inserting software prefetches that may hurt performance.

A cost-benefit analysis filters out delinquent loads that do not benefit from software prefetching. Assuming that a prefetch instruction costs α cycles, $MR_{A\,(D\$)}$ is the miss ratio of load A at the first level cache, and $latency$ is the average latency experienced by each L1 cache miss – the decision to insert a prefetch for delinquent load A is defined by the relation

$$MR_{A\,(D\$)} > \frac{\alpha}{latency}$$

Per-instruction miss ratios for all cache levels are provided by the cache model, and average latency per cache miss can be measured using performance counters. Using ineffective prefetches we found that a prefetch instruction takes 1 cycle. A load is considered favorable for prefetching only if it passes this test. Otherwise, it is not considered for prefetching.

We call our method *model driven delinquent load identification* (MDDLI). The delinquent load identification pass performs the cost-benefit analysis and selects only those delinquent loads for which software prefetching is expected to benefit performance.

VI. Prefetching Analysis

This section describes how prefetches are inserted for delinquent loads identified in the MDDLI pass. Of all the identified loads, only those with regular strides are considered in the prefetching analysis. Our method uses per-instruction stride samples to identify delinquent loads with regular strides. The stride analysis groups all strides of similar size that are likely to fall in the same cache line. After grouping similar strides, the analysis categorizes a load as having a regular stride if more than 70% of its stride samples fall in one stride-group. The analysis then selects the most frequent stride in the dominant stride-group to compute a suitable *prefetch distance*.

A. Prefetch Distance

To effectively hide memory latency, prefetches should be issued a suitable number of iterations earlier than the demand load. The prefetch distance (P) in bytes (number of bytes to prefetch ahead) can be computed with the selected stride as $P = \lceil \frac{l}{d} \rceil \times stride$ [12], where l is the average memory latency[3] and d is the number of cycles it takes to execute one iteration of the loop. We approximate d using recurrence (r) and the *average cycles per memory-operation* (Δ), as $d = (r \cdot \Delta)$. We can measure Δ for each benchmark using performance counters or simply use a constant, such as the average cycles

[1]Stack distance is the number of unique cache lines accessed in-between a cache line reuse

[2]Interested readers can read more about the modeling details in [4]

[3]Average memory latency is required by the cost-benefit analysis (Section V) and is known at this point in analysis

per memory operation across all benchmarks. For our analysis we measured the average cycles per memory operation for each benchmark.

When the stride is smaller than the cache line size (C), the cache line is reused by the factor $i = \frac{C}{stride}$ on average. So we shorten the prefetch distance proportionally. In this case the prefetch distance is computed as:

$$P = \left\lceil \frac{latency}{d \cdot i} \right\rceil \times C$$

For total number of references R in a loop, the first P will be misses. To avoid too many misses, our prefetch distance analysis ensures the relation $P \leq \lfloor \frac{R}{2} \rfloor$.

B. Cache Bypassing

Modern processors support a special data prefetching mechanism called non-temporal prefetch. The non-temporal prefetch instruction (PREFETCHNTA) can be used to prefetch data into the L1 cache without polluting the higher level caches. When this cache line is evicted, it goes directly to the DRAM instead of working its way up the cache hierarchy. This instruction is extremely useful when data is known to *not* be reused temporally from higher level caches, as it helps retain other (possibly) useful data in the higher level caches longer.

To improve our prefetching algorithm with cache-bypassing, we included an analysis (originally proposed by Sandberg et al. [17]) to discover opportunities of cache bypassing. After a load is identified as prefetchable, the analysis identifies the loads that access the same cache line directly after, called the *data-reusing* loads. Data reuse samples reveal the data flow between the sampled loads. The analysis then uses per-instruction miss ratio curve (Figure 3) of the data-reusing loads in the graph to determine if they reuse data from higher level caches. If a load does not reuse data from the L2 and LLC cache, the miss ratio curve will not drop between the points L1$ and LLC (Figure 3). If none of the data-reusing loads reuse data from higher level caches, the prefetchable load can be safely marked as a non-temporal memory access, and we use the PREFETCHNTA instead of an ordinary prefetch instruction. This cache bypassing method is most beneficial when several applications execute in parallel on different cores and share LLC and off-chip bandwidth. It avoids polluting the LLC and lowers off-chip bandwidth use as temporally useful data is retained longer in higher level caches and reused from there. This relief in shared resource demand results in multicore throughput performance benefits. We evaluate software prefetching with and without cache bypassing, for performance in Section VII.

C. Prefetch Insertion

The prefetching analysis pass identifies the load instructions where a software prefetch should be inserted, the type of prefetch (normal or non-temporal) and the ideal prefetch distance. x86 architectures support the *base+offset* addressing mode and inserting a single prefetch instruction of the format "*prefetch offset*(*base*)" suffices. For a load at address A using base register *base* to access memory, the prefetch is inserted right after it in the source as follows

Benchmark	MDDLI filtered		Stride-centric	
	Miss Cov.	OH	Miss Cov.	OH
gcc	65.7%	6.1	63.1%	6.8
libquantum	99.9%	4.9	99.9%	4.9
lbm	98.5%	2.1	98.5%	2.3
mcf	36.4%	1.5	31.2%	2.0
omnetpp	9.0%	5.4	4.1%	13.5
soplex	53.2%	5.1	24.9%	8.4
astar	26.0%	9.6	19.8%	6.9
xalan	3.0%	73.2	1.7%	125.4
leslie3d	93.9%	10.1	97.5%	12.1
GemsFDTD	84.1%	7.7	85.6%	10.1
milc	95.9%	7.2	52.8%	13.0
cigar	28.2%	3.4	28.2%	3.4
Average	58%	11.3	51.1%	17.4

Table I: Prefetch Coverage & Minimization - Our approach executes 35% lesser prefetch instructions than *stride-centric*

```
A: load (base), dst
   prefetch[nta] prefetch-distance(base)
```

The offset here is the computed prefetch distance. The base register is taken directly from the target load. Such optimizations can be applied directly at the binary level via dynamic binary rewriting. Our method is aimed towards compiler and source independence, such that optimizations may be applied even when the source is not available. For simplicity, our framework automatically inserts the optimizations at the assembler level.

D. Prefetch Coverage

In this section we evaluate the how effectively cache misses are covered by our method. Prior works such as [10] and [21] investigated stride profiling to insert software prefetches (Section VIII). These approaches used simple heuristics to determine prefetch insertion. To compare with them, we implemented a profile-guided prefetching method using Pin that inserts prefetches for all loads with regular strides, similar to the methods described. We call this method *stride-centric* and evaluate it alongside ours. From SPEC CPU 2006 we found 11 benchmarks with non-negligible off-chip traffic to evaluate multicore performance during shared resource contention. Here we also study the open source genetic algorithm *cigar* [6]. We applied both our *MDDLI filtered stride analysis* as well as the *stride-centric* method to these benchmarks. Table I shows the coverage of both these methods for the 12 benchmarks when compared to functional simulation (Section IV). The average L1 miss coverage across all benchmarks is 58%. Miss coverage is lowest for *omnetpp* and *xalan*. For *omnetpp*, the delinquent loads identified by MDDLI cover 89% misses. However, there is very little opportunity for stride prefetching (due to irregular pointer chasing), resulting in only 9% miss coverage after applying our stride prefetching algorithm. *xalan* suffers from lower miss coverage from MDDLI (63%), as well as limited opportunity for stride prefetching. The column titled *OH* (overhead) shows the number of prefetch instructions executed per one miss removed. Compared to our method, the stride-centric method executes 36% more prefetches on average. Our method's cost-benefit analysis minimizes the prefetch instructions inserted. We evaluate the performance of *stride-centric* method in the next section alongside our software prefetching method.

VII. Evaluation

In this section we evaluate the performance of the two software prefetching techniques, with and without cache bypassing, on two high performance x86 multicore processors listed in Table II. We optimized for both target architectures using a single input profile. Here we also compare the performance of the hardware prefetcher on both processors when running the original benchmarks.

CPU	L1$	L2$	LLC	Freq.
AMD Phenom II	64 kB	512 kB	6 MB	2.8 GHz
Intel i7-2600K	32 kB	256 kB	8 MB	3.4 GHz

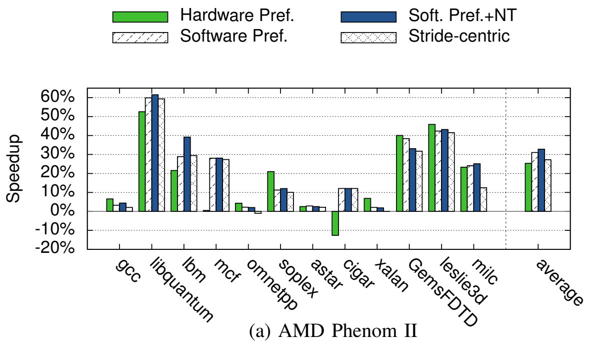
Table II: Processor details

The 12 benchmarks evaluated for performance are listed in Table I. As discussed earlier, our framework works at the assembler level and is not dependent on any particular compiler. For our experiments we used GCC 4.7 and compiled the original benchmarks with O2 optimization. The assembler output for the same programs was used by our framework to automatically schedule prefetch instructions. For all our evaluations the baseline is the performance of the original benchmarks, with hardware prefetching turned off. This arrangement is important to isolate the speedup and off-chip traffic performance of our method.

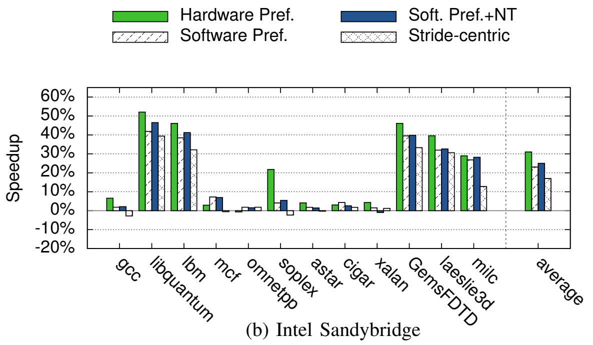
A. Speedup

Figure 4 compares the speedup from our prefetching methods (with cache bypassing "*Soft.Pref.+NT*" and without cache bypassing "*Software Pref.*"), *stride-centric* method, and from hardware prefetching, when the benchmarks are run in isolation. Significant speedups are observed for our software prefetching method on both processors for several benchmarks such as *libquantum* (up to 62%), *lbm* (up to 41%), *mcf* (up to 28%), *cigar* (up to 13%), *leslie3d* (up to 42%) and *GemsFDTD* (up to 40%). Cache bypassing ("*Soft.Pref.+NT*") further improves performance considerably for *gcc*, *libquantum*, *lbm*, *soplex* and *leslie3d* on AMD Phenom II with the addition of *GemsFDTD* on the Intel processor. Our prefetching strategy outperforms hardware prefetching on AMD for *libquantum*, *lbm*, *mcf*, *astar* and *cigar* and on Intel for *mcf* and *cigar*. Hardware prefetching on AMD slows down *cigar* by more than 11% as it's short lasting strided accesses trick the stride prefetcher in to prefetching useless data, which removes useful data from the cache and hurts performance. Hardware prefetching on Intel benefits *cigar* performance because of prefetching adjacent cache lines on a miss. Compared to hardware prefetching, our method posts smallest relative improvement for *soplex* on both processors (its miss coverage is only 53% - Table I). While experimenting with our prefetching methods, we observed that prefetches from L2 alone improved *libquantum* by 4%, *lbm* by 3% and *soplex* by 1.3% on AMD. *Stride-centric* performs worse than our software prefetching on both processors, while fetching considerably more data from the DRAM on several benchmarks (Figure 5).

Overall, our software prefetching (with cache bypassing) comes out ahead of plain software prefetching (without cache bypassing) and far ahead of the stride-centric prefetching. On average, it performs considerably better than hardware prefetching on AMD. On Intel it performs within 5% of hardware prefetching with significantly lesser off-chip traffic (Section VII-B).



(a) AMD Phenom II



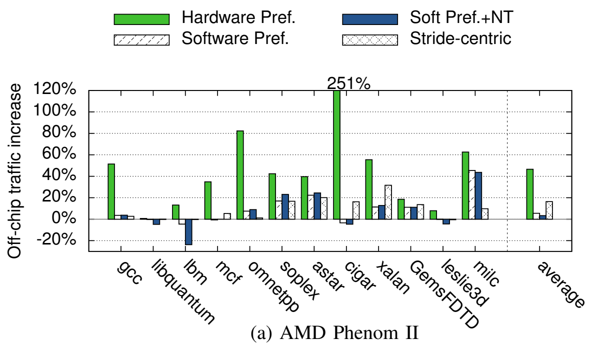
(b) Intel Sandybridge

Figure 4: Speedup of selected benchmarks with different prefetching policies

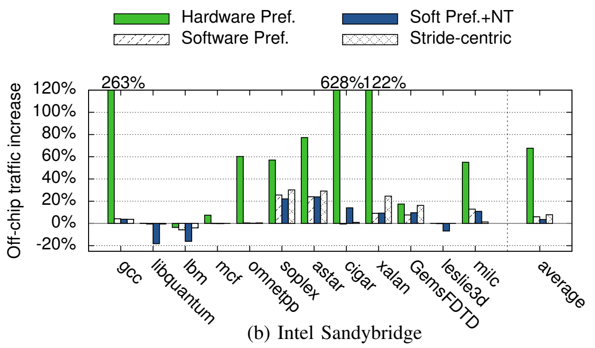
B. Off-chip Traffic

Figure 5 compares the increase in volume of data fetched (compared to baseline) from the DRAM during the entire execution of the benchmarks[4]. An increase in off-chip traffic indicates waste of shared LLC space and off-chip bandwidth. This is critical for the performance of threads that co-execute on neighboring cores and share these resources. Our prefetching strategy proves to be *strictly* better than hardware prefetching on both AMD and Intel. Hardware prefetching is noticeably more aggressive and fetches considerably more data than our proposed software prefetching method. Intel's hardware prefetcher performs especially bad on *cigar* increasing the off-chip traffic by 630% compared to the baseline. Cache bypassing significantly reduces the off-chip traffic over the baseline on both processors (up to 22%) for *libquantum*, *lbm*, and *leslie3d*. This is because useful data is retained and reused from higher level caches for a longer period instead of being evicted and re-fetched from the DRAM. On average, our prefetching method (with cache bypassing) lowers off-chip traffic by 44% on AMD and 64% on Intel compared to hardware prefetching, while maintaining comparable performance. Reduced off-chip traffic lowers off-chip bandwidth demand and reduces LLC pollution. This benefits throughput performance when several cores become active (Section VII-C).

[4]Measurements were made using performance counters

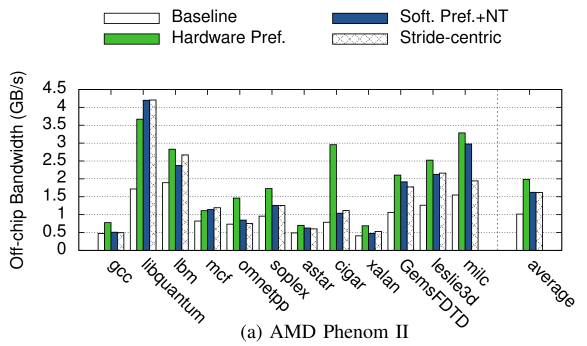


(a) AMD Phenom II

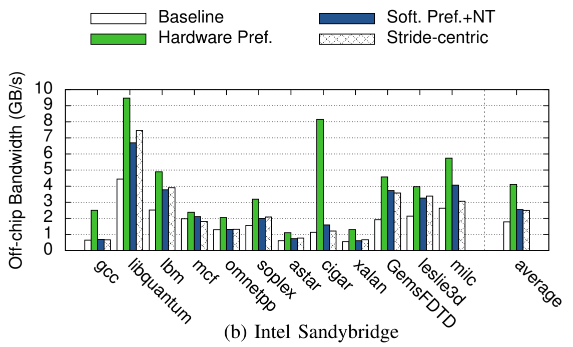


(b) Intel Sandybridge

Figure 5: Increase in Data Volume fetched from the DRAM over the lifetime of benchmarks



(a) AMD Phenom II



(b) Intel Sandybridge

Figure 6: Average Memory Bandwidth (Gigabytes per second) for benchmarks

Figure 6 shows the average off-chip bandwidth consumed by the benchmarks. Our software prefetching consumes significantly less bandwidth on both platforms, with the exception of *libquantum* on AMD Phenom II (where it also significantly outperforms hardware prefetcher's performance). On average, our prefetching approach (with cache bypassing) lowers bandwidth consumption by 19% on AMD and 38% on Intel, compared to hardware prefetching. The reduction in the off-chip bandwidth consumption is proportional to the reduced off-chip traffic. We don't report bandwidth usage for our simple software prefetching (without cache-bypassing) since it is similar to the one with cache bypassing.

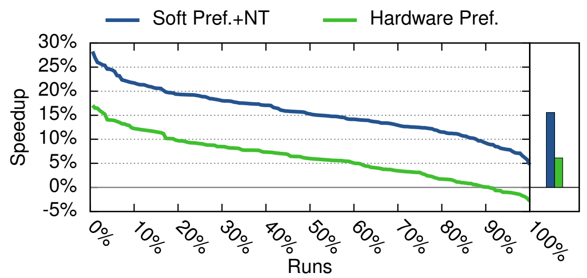
C. Mixed Workloads

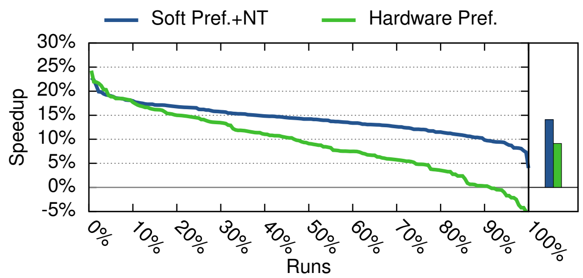
Hardware prefetching significantly increases off-chip traffic, thereby frequently polluting the shared last level cache, and consuming more off-chip bandwidth. This directly impacts the performance of threads co-executing on neighboring cores. To assess the impact of our method's resource conservation in multicores, we ran 180 randomly generated workload mixes on both processors. Each mix contains four randomly selected workloads that are run in parallel on four cores[5]. We compare the impact on throughput performance (weighted speedup) and shared-resource pollution (increased off-chip traffic) when using our software prefetching method and hardware prefetching alone in Figure 7. The baseline is the original mix with hardware prefetching turned off. The graphs (Figure 7) show the distribution function of throughput performance and off-chip traffic increase across the 180 mixes[6]. For example, Figure 7a shows that in 60% of the mixes, our method improves throughput by at least 14%, whereas hardware prefetching improves throughput by 5% or more. Similarly, Figure 7c shows that 60% of the time our method increases off-chip traffic by at most 20%, while hardware prefetching increases by up to 28%.

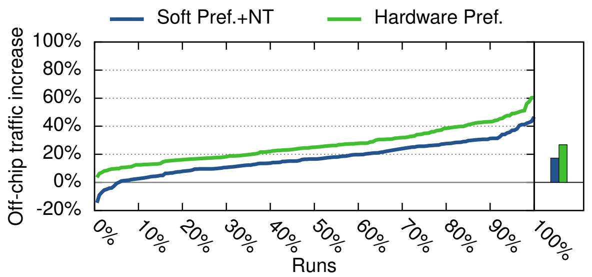
On Intel our software prefetching method performs about 5% better than hardware prefetching on average. Hardware prefetching slows down 9% of the mixes over the baseline. Our prefetching mechanism gets minimum speedup of 5% and achieves higher throughput than hardware prefetching in 143 (79%) of the 180 mixes. Software prefetching performs strictly better in terms of off-chip traffic increase in all cases (Figure 7d). 73% of the mixes have lower off-chip traffic than the baseline due to effective cache bypassing. On average, off-chip traffic is reduced by 3% over the baseline and 28% over hardware prefetching. The significant increase in off-chip traffic for hardware prefetching reveals the high likelihood of cache pollution when running low on shared LLC space.

[5]benchmarks that run longer than others, experience lesser shared resource contention

[6]The graphs are sorted for both, the hardware prefetching and our prefetching approach.


(a) Speedup on AMD (higher is better)


(b) Speedup on Intel (higher is better)


(c) Off-chip Traffic Increase on AMD (lower is better)

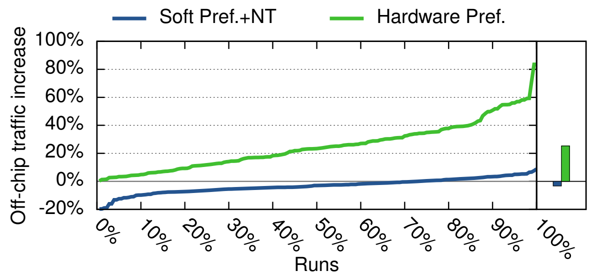

(d) Off-chip Traffic Increase on Intel (lower is better)

Figure 7: Distribution function of performance across 180 mixed workloads on AMD and Intel processors (averages on right)

Figure 8 explores a single workload mix in detail. It shows the speedup of the individual benchmarks (over their baselines) in the mix on Intel. The mix contains the benchmarks *cigar, gcc, lbm* and *libquantum*. Figure 4b shows that in single threaded runs, all four benchmarks perform better with hardware prefetching compared to our prefetching method. However, they also consume significantly more off-chip bandwidth with hardware prefetching. Based on their single-threaded off-chip bandwidth requirements with hardware prefetching (Figure 6b), the mix would require a maximum bandwidth of 25.3 GB/s. However, due to aggressive hardware prefetcher requests the mix only achieves a maximum off-chip bandwidth of 13.6 GB/s (Figure 8), saturating bandwidth resources much earlier and slowing down the mix. Our software prefetching method on the other hand ideally requires (12.8 GB/s) and consumes significantly lesser off-chip bandwidth (10 GB/s), which results in 20% higher throughput than hardware prefetching.

On AMD Phenom II our software prefetch approach achieves higher throughput than hardware prefetcher's best performance in 41% of the cases. Our software prefetching method performs strictly better than hardware prefetching for all 180 cases, increasing throughput by 16% on average. We observe a maximum throughput improvement of 24% over hardware prefetching. Hardware prefetching improves throughput by only 6% on average, and degrades performance for 9% of the mixes over the baseline. Whereas our software prefetching method never hurts performance and improves throughput by at least 5%. Compared to hardware prefetching, this approach reduces off-chip traffic by up to 25%, and 10% on average (Figure 7c).

We conclude that hardware prefetching can be too aggressive when off-chip bandwidth is limited, and it does not benefit overall performance as much as a resource efficient prefetching method can. Our method's increased throughput as a result of shared-resource conservation (reduced off-chip traffic) highlights the importance of accurate use of shared resources.

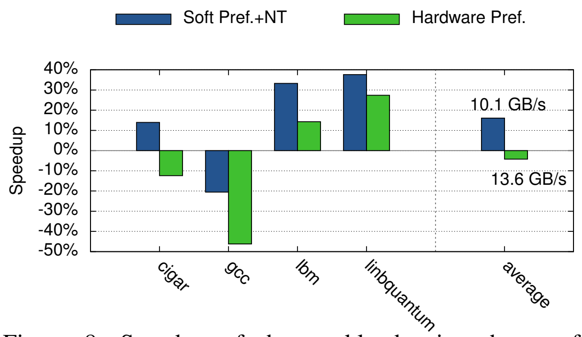
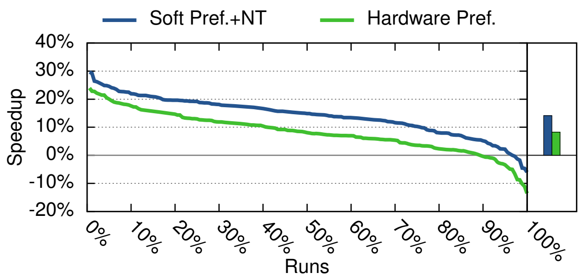


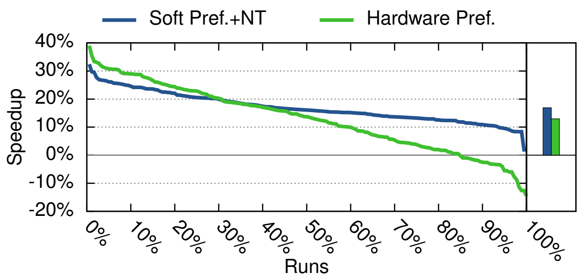
Figure 8: Speedup of the workload mix where software prefetching has the largest benefit over hardware prefetching in Intel. The maximum off-chip bandwidth for the mix is written next to the averages.

D. Varying Inputs

To determine the sensitivity of our prefetching method to varying input sets, we ran the mixed workloads with input sets other than those used during sampling. Benchmarks with several alternate input sets (for example *gcc*) were run with a randomly selected input for each mix. We varied the inputs across all mixes. Figure 9 shows the speedup for the 180 workload mixes over the baseline for both our prefetching method and hardware prefetching. Our approach performs 6%



(a) AMD Phenom II



(b) Intel Sandybridge

Figure 9: Distribution function of Speedup across 180 Mixed workloads using different inputs (higher is better)

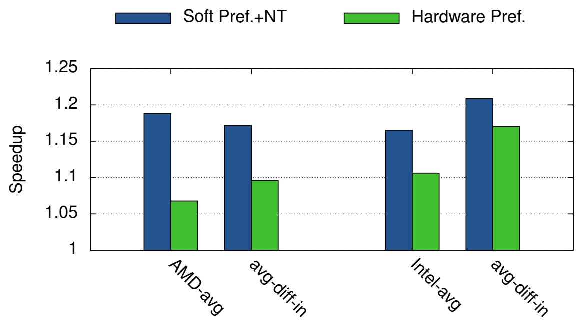


Figure 10: Fair-Speedup (normalized to baseline) - average for 180 Mixed workloads on AMD and Intel. Averages for the runs using the original and different inputs is shown.

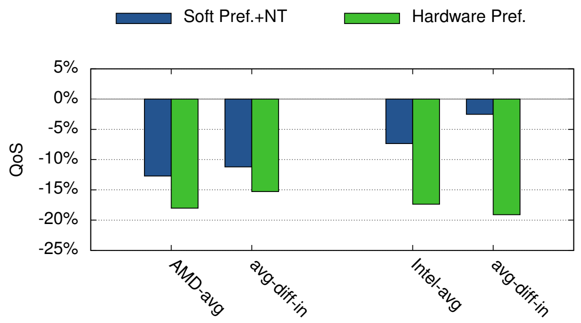


Figure 11: QoS degradation - average for 180 Mixed workloads on AMD and Intel. Closer to zero indicates lesser cumulative slowdown in the mixes. Averages for the runs using the original and different inputs is shown.

better on average than hardware prefetching on AMD Phenom II (Figure 9a) while fetching 12% less data. The average speedup over hardware prefetching is 4% on Intel (Figure 9b) with 75% lower off-chip traffic. 95% of the mixes speed up by at least 10% over the baseline. We observe that our method's performance is more stable (irrespective of input) compared to hardware prefetching. Our method mostly improves throughput between 10% to 30%, whereas hardware prefetcher's performance varies significantly and degrades throughput for 10% of the mixes. We conclude that our resource-efficient method is stable across different inputs and as a result of shared resource conservation it significantly outperforms hardware prefetching.

We report additional mixed workload performance metrics (similar to those used in [20]) to further elaborate our prefetching method's efficiency

- *Fair-Speedup* (FS) - It balances both fairness and speedup and is computed as the Harmonic mean of the per-application speedup in the mixes:

$$FS = N / \sum_{i=1}^{4} \frac{Exec\text{-}Time_{app_i(prefetching)}}{Exec\text{-}Time_{app_i(base)}}$$

Figure 10 shows the average FS across the 180 mixes. The performance benefit reported by FS for our software prefetching is similar to weighted speedup (shown in Figure 7 and Figure 9). Our resource efficient prefetching performs significantly better than hardware prefetching.

- *Quality of Service* (QoS) - Cumulative application slowdown per workload mix, computed as

$$QoS = \sum_{i=1}^{4} min\left(0, \frac{Exec\text{-}Time_{app_i(base)}}{Exec\text{-}Time_{app_i(prefetching)}} - 1\right)$$

Figure 11 shows how QoS is negatively impacted on average across the 180 mixes. Ideally it should be zero implying that no application in any mix ever slowed down. Our software prefetching method ensures significantly less QoS degradation than hardware prefetching. It is interesting to see that the impact on QoS for our software prefetching method lowers (i.e. QoS improves) when going from the original to different inputs. This is because prefetching in general causes an imbalance in resource sharing (compared to the baseline) as a result of which some applications in the mix may slowdown (Figure 8) resulting in the reported QoS degradation. When the input set is changed, the software prefetching is not optimal, resulting in smaller performance benefits compared to original inputs (Figure 10). However, on a positive note the non-optimal prefetching also causes lesser imbalance in resource sharing between the applications, which results in smaller QoS degradation.

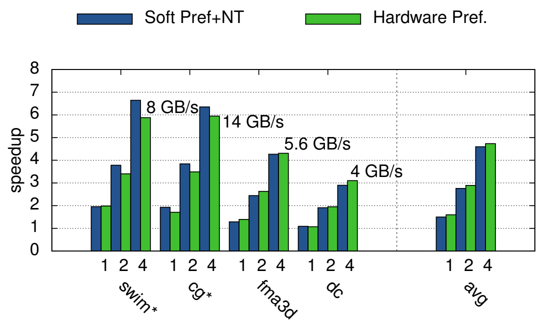


Figure 12: Speedup of four parallel workloads for 1, 2 and 4 threads on Intel. *swim* and *cg* have the highest off-chip bandwidth (shown) in the SpecOMP and NAS suites. *swim* uses only half of the total available bandwidth.

E. Multi-threaded Workloads

We also investigated performance across the NAS and Spec OpenMP parallel benchmarks. Most of the benchmarks are computationally bound and do not saturate the off-chip bandwidth, therefore the hardware prefetcher does a perfect job at prefetching the required data. For the Intel processor, *streams* benchmark reported maximum off-chip bandwidth of 15.6 GB/s, which was not fully achieved by any parallel benchmark even with 4 threads. Due to this our software prefetching approach mostly has performance comparable to the hardware prefetcher. Figure 12 compares the performance of four parallel benchmarks, two with the highest off-chip bandwidth (marked *) and two ordinary cases. Our prefetching approach shows performance benefit over hardware prefetching when demand for offchip-bandwidth is higher, and has comparable performance in the ordinary cases, and on average across all NAS and Spec OpenMP benchmarks. We observed similar performance behavior on AMD. Since shared resources are more readily contended when running mixed workloads, resource-efficient prefetching is less interesting for streaming parallel workloads.

VIII. Related Work

This paper makes contributions to the areas of i) low-overhead delinquent load identification, and ii) Software prefetching. We describe the relevant related work for both parts separately.

A. Delinquent Load Identification

Nearly all software prefetching techniques use some kind of delinquent load identification method to guide software prefetch insertion. Barnes et al. [14] and Rabbah et al. [15] used cache simulations to identify delinquent loads. Simulations are prohibitively slow, so we proposed the use of low-overhead cache modeling to identify delinquent loads. Panit et al. [13] combined analytical models with profiling to improve the accuracy of delinquent load identification, avoiding simulation. The overhead for profiling even with training data can be several times slower than native execution.

Zhao et al. [23] proposed an online prefetching method that uses mini-simulations of short memory address traces to identify delinquent loads responsible for up to 69% of misses (in a 512 kB L2 cache) on average. Our method reports 94% miss coverage for the same sized cache. They report an online overhead of 14%. This technique requires separate thread contexts and spare shared resources for enabling online simulations. Such a method is not optimal for increasing throughput of highly utilized processors.

Some frameworks such as [1], [7], [9] use extensive performance counters and architectural registers support in Intel Itanium processors to sample long latency memory access events to identify delinquent loads. Although most of the used events are available in PEBS on all modern Intel processors, such frameworks are generally not portable to other platforms. Other approaches such as [2], [3], [22] propose hardware extensions to quickly identify delinquent loads for guiding software prefetching. Our method does not require any special hardware features and works on real hardware.

B. Software Prefetching

Existing work on software prefetching can be categorized into i) Static analysis based, ii) Profile Guided, and iii) Online Profile Guided. We describe the relevant related work for them separately.

1) Static Compile Time Analysis: Moore et al. [12] and Santhanam et al. [18] discuss the implementation of compiler based software prefetching algorithms and show their effectiveness for scientific workloads. Implementation of similar software prefetching algorithms are available in production compilers such as GCC and Open64. However, software prefetching guided by static compiler analysis is shown to degrade single-thread performance [11]. Lee et al. [5] observed negligible speedups with GCC profile guided optimization (PGO), the improvements mainly attributed to optimizations other than software prefetching. They also observed that Intel's *icc* compiler (with PGO) does not insert any software prefetches. We observed the same with the *icc 12.4* compiler. The scope of static analysis based prefetching is limited by that the stride needs to be known at compile time, limiting other prefetching opportunities, such as pointer chasing in regular data structures.

2) Profile Guided: Wu et al. [21] complemented edge-profiling with a stride profiling method to discover loads with frequently recurring strides. The stride profiles are used to guide software prefetch insertion into the source code. Luk et al. [10] developed a profiling method to identify loads exhibiting regular strides during execution. However, both these approaches use simple heuristics to identify what loads to prefetch for. We have compared a *stride-centric* approach with our proposal. Our method schedules lesser prefetch instructions and clearly performs better. Moreover, the profiling overhead for both these approaches is very high compared to our method. Luk et al. [10] reports average overheads between 7×-15×.

Lee et al. [5] studied the combined affect of software prefetching and hardware prefetching on performance. Our experiments combining hardware and software prefetching confirmed their observation that combining the two can hurt performance in several cases and should be avoided.

3) Online Profile Guided: Dynamic optimization frameworks such as [1], [7] use extensive (Itanium-specific) architectural register support for sampling relevant memory events and performance counters to detect delinquent loads to prefetch data for them. Beyler et al. [1] monitor stride behavior of regularly occurring loads at runtime using a separate thread context. Once the monitoring thread notices a regular stride, it inserts software prefetch for that load on the fly. However, this approach degrades performance for several benchmarks when the prefetching can not amortize the cost of the runtime system's overhead. Such techniques are also not portable to other platforms. In contrast, our architecture-independent approach can target several cache configurations for different processors in a single analysis pass and does not require spare resources for separate thread contexts.

To the best of our knowledge, this is the first work to demonstrate the advantage of using a shared resource-conserving software prefetching method over aggressive hardware prefetching in commodity multicores.

IX. Conclusions

This work investigates how a resource-efficient prefetching method can help improve throughput performance in multicores when shared resources are constrained. We proposed an efficient method that 1) accurately prefetches the required data, 2) avoids (useless) speculative prefetching, and 3) employs cache bypassing to retain useful data in the higher level caches. In contrast to hardware prefetchers (in commodity multicores), this resource-efficient prefetching method is designed to maintain minimal off-chip traffic, and as a result avoids LLC pollution and lowers off-chip bandwidth demand. This helps throughput performance in multicores when several applications co-execute and share resources. We thoroughly demonstrated this benefit on two modern multicores with 180 mixed workloads that fully utilized the processors. Our results showed that both multicores mostly achieved higher throughput (and lesser off-chip bandwidth) when using resource-efficient prefetching compared to hardware prefetching. Compared to state-of-the-art hardware prefetching on two processors, our resource efficient prefetching method performed 10% and 5% better on average. Our work highlights the importance of shared-resource friendly prefetching for optimizing performance in multicores.

References

[1] J. C. Beyler and P. Clauss. Performance Driven Data Cache Prefetching in a Dynamic Software Optimization System. In *ICS*, 2007.

[2] 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 *ISCA*, 2001.

[3] R. Cooksey, S. Jourdan, and D. Grunwald. A Stateless, Content-Directed Data Prefetching Mechanism. In *ASPLOS*, 2002.

[4] D. Eklov and E. Hagersten. StatStack: Efficient Modeling of LRU caches. In *ISPASS*, 2010.

[5] J. Lee, H. Kim, and R. Vuduc. When Prefetching Works, When It Doesn't, and Why. *ACM TACO*, 9(1), Mar. 2012.

[6] S. J. Louis. CIGAR - Case Injected Genetic Algortihm. http://www.cse.unr.edu/ sushil/class/gas/code/cigar/.

[7] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *MICRO*, 2003.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[9] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *CGO*, 2004.

[10] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-Guided Post-Link Stride Prefetching. In *ICS*, 2002.

[11] J. Mars and R. Hundt. Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations. In *CGO*, pages 169–179, 2009.

[12] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS*, 1992.

[13] V.-M. Panait, A. Sasturkar, and W.-F. Wong. Static Identification of Delinquent Loads. In *CGO*, 2004.

[14] R. C. R. Barnes and D. Gillies. Feedback-Directed Data Cache Optimizations for the x86. In *Proc. ACM Workshop on Feedback-Directed Optimizations*, 1999.

[15] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler Orchestrated Prefetching via Speculation and Predication. In *ASPLOS*, 2004.

[16] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *ISCA*, 2009.

[17] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *SC*, 2010.

[18] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data Prefetching on the HP PA-8000. In *ISCA*, 1997.

[19] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase Guided Profiling for Fast Cache Modeling. In *CGO*, 2012.

[20] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *SC*, 2009.

[21] Y. Wu. Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching. In *PLDI*, 2002.

[22] W. Zhang, B. Calder, and D. M. Tullsen. A Self-Repairing Prefetcher in an Event-Driven Dynamic Optimization Framework. In *CGO*, 2006.

[23] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous Memory Introspection. In *CGO*, 2007.