Characterizing the Impact of Prefetching on Scientific Application Performance

Collin McCurdy
Future Technologies Group
Oak Ridge National Laboratory
Email: cmccurdy@ornl.gov

Gabriel Marin
Innovative Computing Laboratory
University of Tennessee
Email: gmarin@utk.edu

Jeffrey Vetter
Future Technologies Group
Oak Ridge National Laboratory
Email: vetter@ornl.gov

***Abstract*—In order to better understand the impact of data prefetching on scientific application performance, this paper introduces two analysis techniques, one *micro-architecture*-centric and the other *application*-centric. We use these techniques to analyze representative full-scale production applications from five important Exascale target areas. We find that despite a great diversity in prefetching effectiveness across and even within applications, there is a strong correlation between regions where prefetching is most needed, due to high levels of memory traffic, and where it is most effective. We also observe that the application-centric analysis can explain many of the differences in prefetching effectiveness observed across the studied applications.**

I. Introduction

Due to power and scaling limitations, the available DRAM per processing core is projected to shrink dramatically (up to a factor 33) by the time systems reach Exascale [11]. While innovations in NVRAM technology (Memristors [8], STT-RAM [13], PCRAM [5]) offer the hope of more dense and thus more bountiful memory, it would likely come at the expense of latency.

Memory parallelism reduces the impact of latency. Data prefetching, that is identifying data that will be needed and moving it closer to the processing unit so that it is available when required by a running application, can increase memory parallelism and is therefore potentially extremely important in this new environment. However, prefetching always carries the danger that ineffective prefetches will degrade performance: data that is either not used, or will not be used in a timely fashion, can remove useful data from caches near the processing unit where it is needed.

Surprisingly, this extremely important, yet potentially dangerous, mechanism is largely hidden from users. Software prefetching instructions are generally inserted by compiler, and prefetching structures are *micro*-architectural features in hardware.

In order to better understand the impact of data prefetching on scientific application performance, this paper introduces two analysis techniques.

The first technique is *micro-architecture*-centric: we use mechanisms that allow disabling of hardware prefetching in a particular hardware implementation (AMD 10H [4]), along with hardware performance counters that accurately measure prefetching events, to systematically probe the space in an existing design, isolating the effects of multiple levels of hardware prefetchers on applications.

The second technique is *application*-centric: we have designed a tool that abstractly simulates hardware mechanisms for stream detection *for an arbitrary number of streams*, allowing us to determine the number of streams active at any one time in an application.

We have used these techniques to analyze representative full-scale production applications from five important Exascale target areas: fusion energy (GTC), climate (CAM-HOMME), molecular dynamics (LAMMPS), materials science (NEK), and combustion (S3D). We have further identified the dominant loop nests in each application, enabling a deeper understanding of underlying reasons for performance.

Overall, despite a great diversity in prefetching effectiveness (as measured by performance improvement over no prefetching) across and even within applications, we see a strong correlation between regions where prefetching is most needed (due to high levels of memory traffic) and where it is most effective.

Additionally, we find:

- While hardware prefetching always improves application performance in the applications we studied, compiler-inserted software prefetches improve performance substantially in some applications, but degrade performance substantially in other applications.
- While a hardware prefetcher at later levels of the memory hierarchy (such as the memory controller) can boost *serial* performance significantly, contention for the shared resource can substantially reduce effectiveness in a parallel context.
- Prefetching substantially increases already high levels of memory parallelism, *even in applications that prominently feature irregular data accesses*.

II. Related Work

There is a great deal of work in the literature regarding both hardware and software prefetching.

Work regarding hardware prefetching tends to focus on new ideas for prefetching structures and implementations. The ideas are implemented in simulators and evaluated using benchmarks, or portions of benchmarks, meant to represent full applications [7], [23], [10], [9].

Literature on software prefetching focuses primarily on algorithms for determining where compilers should automatically insert prefetch instructions, again evaluated using benchmarks though often running on actual hardware [6], [22], [17].

There is little work in the literature evaluating the impact of *existing* micro-architectural and compiler prefetching implementations on full-scale application performance.

As regards our methodology, while there is plenty of discussion on the World Wide Web about mechanisms for disabling prefetching, and even small studies that benchmark performance of embedded systems kernels with and without prefetching hardware enabled [1], to our knowledge this is the first description of work that systematically enables and disables multiple levels of prefetching mechanisms to isolate their impact on full-scale application performance.

Previous work has used simulators [21] and models [15] to understand the performance limits of prefetching. A description of our approach for characterizing the number of inherent streams in an application, technique developed as part of this study, has been expanded and published separately in [18] before the publishing of this work.

III. Prefetching Hardware and Software

Prefetching is a very effective technique for hiding memory latency and increasing application performance. Prefetching works by eagerly loading into the cache data that is expected to be needed in the near future. To be most effective, the data must be fetched sufficiently far in advance that its loading is completed by the time the micro-processor needs it. At least as important, prefetching must predict correctly the data that must be fetched in advance. Incorrect predictions increase demand on the memory subsystem, possibly evicting useful data from caches, and increase bandwidth use.

Prefetching comes in two main flavors: software prefetching and hardware prefetching. Software prefetching is the more common type of prefetching. Compilers generally, but also programmers, insert explicit prefetch instructions in the code to fetch data that will be needed in the near future. Prefetch instructions are similar to load instructions, except that they do not create a dependence on the loaded data. In addition, temporal hints can be associated with prefetch instruction to indicate the cache level where data should be loaded.

Hardware prefetching works without any support from the compiler or the programmer. Prefetchers based on Jouppi's stream buffers [14] are nowadays commonly found in modern micro-processors. The hardware detects easy to recognize access patterns, such as strided memory accesses, and speculatively fetches the memory addresses predicted to be accessed in the near future. While a compiler may perform more expensive analysis on the program code and can understand more complex memory access patterns, the hardware can observe the stream of dynamic addresses which may be regular even if the access pattern cannot be statically predicted. Thus, each one of the two approaches can perform better in different situations.

In rare cases, software prefetching can hurt performance due to increased issue demand on the load/store units. The hardware prefetcher works outside the core, and thus it does not affect issue bandwidth. However, both prefetch approaches can increase memory bandwidth demand. In the following section we look more closely at the two hardware prefetchers of the AMD 10H micro-architecture, and we perform an empirical evaluation of their performance.

A. AMD Hardware Prefetchers

While the topic of hardware prefetching has been extensively studied in literature, few details have been disclosed about the actual implementations used by AMD microprocessors. We know that the AMD Shanghai and the AMD Istanbul micro-architectures incorporate two levels of hardware prefetching. The first prefetcher is associated with the data cache level and is replicated across all the cores of a microprocessor. We call this the DC prefetcher. The DC prefetcher analyzes the stream of memory addresses generated by data cache misses and attempts to predict addresses that will be accessed by the CPU core in the near future. If a predicted memory location is not already in the data cache, the prefetcher fetches that data from L2, L3 or from DRAM. Because the prefetched data is brought into the data cache, the prefetcher operates at cache line size granularity.

The second prefetcher, the MC prefetcher, is associated with the memory controller. It operates on the stream of memory addresses produced by accesses that miss in the last level of cache and that originate from any of the cores connected to that memory controller. The MC prefetcher fetches data into a separate prefetch buffer to avoid conflicts with data loaded into the CPU caches.

The prefetchers operate by recognizing strided memory accesses. When a prefetcher recognizes an address stream, it stores the stream information into an internal data structure, and fetches the location that is predicted to be accessed next. Such streaming prefetchers can track multiple data streams at the same time, and the prefetch distance, how many predicted accesses ahead to prefetch, can be adjusted based on the observed behavior.

The prefetchers' performance is affected by several design choices: 1) the *maximum stride* represents the largest streaming stride that can be detected by the hardware; 2) the *stream table size* determines how many distinct streams can be tracked concurrently; 3) the *associativity of the stream table* determines the probability of getting conflicts in the stream table; 4) the *prefetch distance*, how many lines in advance to prefetch, determines if the prefetched data is fully loaded into the cache by the time the application needs it.

Based on data published by AMD [3], the DC prefetcher can fetch data up to 3 lines in advance, while the MC prefetcher fetches data up to 5 lines in advance. We analyze the other characteristics empirically. To understand the prefetchers' characteristics, we developed a micro-benchmark to observe the effects on performance while we probe part of the design space. The micro-benchmark executes streaming memory accesses with a configurable behavior. Command line arguments control the number of concurrent streams, the stride between consecutive accesses to the same stream, the total size of the memory block used by the benchmark, the number of memory access to be performed, and the data alignment between different streams. In addition, the micro-benchmark

SET	Event 1	Event 2	Event 3	Event 4
BASE	cycles	inst	sse_ops	L1_acc
L1	L1_acc	L1_miss	L1_frL2	L1_frNB
L2	L2_req_DC	L2_miss	L2_fill	L2_evict
L2REQ	L2_req	L2_req_DC	L2_req_PF	L2_req_SNP
L3	L3_req	L3_miss	L3_fill	L3_evict
MC	MemCtl	MemCtlWr	MemCtlRd	MemCtlPf
NODE	Rd_Req0	Rd_Req1	DRAM_Req0	DRAM_Req1
TLB	L2TLB_hit	TLB_miss	L2_req_TLB	L2m_TLB
PF	L2_req_PF	L2m_pf	MemCtlPf	sw_pf_ALL
SWPF	sw_pf_LS	sw_pf_NTA	ineff_L1	ineff_L2

TABLE I. EVENT SETS USED IN THE STUDY. NOTE THAT THERE IS SOME DUPLICATION OF EVENTS BETWEEN SETS, ALLOWING SANITY CHECKS. ADDITIONALLY, EACH EVENT SET IMPLICITLY INCLUDES CYCLES (AS READ FROM THE TIME STAMP COUNTER REGISTER VIA THE RDTSC INSTRUCTION).

Configuration Name	Mem Size (in MB)	# of Streams	Stream Size (in pages)	Stride (in lines)
MaxStride	96	1	8*k+1	-8 to +8
Align 1	32	1 – 24	16*k+1	1
Align 2	32	1 – 24	16*k+2	1
Align 4	32	1 – 24	16*k+4	1
Align 8	32	1 – 24	16*k+8	1
Align 16	32	1 – 24	16*k+0	1
Parallel	32	1 – 24	8*k+1	1

TABLE II. BENCHMARK CONFIGURATIONS USED IN THE EMPIRICAL EVALUATION.

uses pointer chasing for all its memory accesses to prevent the compiler from inserting prefetch instructions. We used hardware counters to measure the sets of performance events `L2REQ` and `MC` shown in Table I, while running the micro-benchmark with different configuration parameters.

B. Empirical Evaluation of the AMD 10H Prefetchers

Table II presents a summary of the parameters used for the experiments in this study. While most parameters are self explanatory, we discuss briefly the values chosen for some of them.

All the streams operate on a contiguous block of memory of size given by the *Mem Size* parameter. This memory block is divided into a number of equally sized chunks, corresponding to the number of streams. The micro-benchmark executes memory accesses from each stream in a round-robin fashion. The total memory block size and the number of streams determine an address alignment among the streams. While we were trying to understand the sizes of the different hardware tables by varying the number of concurrent streams in our micro-benchmark, we were seeing a lot of unexplained performance variation in our measurements. Hence, we concluded that some of the hardware data structures must have limited associativity and what we were observing were access conflicts.

To understand these effects, we added the *Stream Size* parameter to control the alignment between two consecutive streams. Based on empirical observations and due to reasons that will be explained in Section III-B2, we try to control the stream alignments at page granularity. We added three parameters to control this alignment. A page factor specifies that the stream size is a multiple of the given number of pages. Thus, $16 * k$ specifies that the stream size is a multiple of 16 pages. Second, a page term specifies how many additional pages are added to the stream size. A stream size of $16 * k + 1$ means that the streams are a multiple of 16 memory pages, and then one more page is added. Using these two parameters we can enforce the page alignment between two streams. For example, a stream size of $16 * k + 2$ specifies that the streams are two page aligned, but they are not 4-page, 8-page or 16-page aligned.

However, even with these alignments we were still observing noise in our measurements, noise that was caused by the 2-way associativity of the L1 cache. The AMD Shanghai and Istanbul micro-architectures have 64KB, 2-way set-associative L1 data caches. Thus, each cache way is 32KB, or 8 pages. By trying to align the streams at 2, 4, 8 and 16 pages, accesses to all streams were hitting in one or a handful of cache sets causing the prefetched data to be evicted before it was used. We were also observing conflict effects on the running times when both prefetchers were disabled. To fix this issue, we added a third parameter which specifies a number of cache lines to be added to the size of each stream. This parameter is not included in Table II, but it was set to 1 for all experiments. By staggering the streams by one cache line, we ensure that consecutive accesses to different streams hit in different L1 sets, while only affecting the desired stream page alignment for accesses near page boundaries.

We often use the terms *prefetch* or *prefetching effectiveness* during this empirical evaluation, to quantify the performance of the hardware prefetchers. Our micro-benchmark generates only clean strided accesses. For this section, we define *prefetching effectiveness* to mean the fraction of data accesses to the next memory level that are initiated by the hardware prefetcher. Thus, for the DC prefetcher, the prefetching effectiveness is computed as

$$DC_{effectiveness} = L2\_req\_PF/L2\_req\_DC,$$

where *L2_req_PF* represents the number of requests to L2 initiated by the DC hardware prefetcher, and *L2_req_DC* represents the number of L2 requests originating from the data cache. Note that the latter event includes both requests to L2 caused by L1 cache misses and requests initiated by the DC prefetcher.

Similarly, the MC prefetching effectiveness is computed as

$$MC_{effectiveness} = MemCtlPf/MemCtlRd,$$

where the *MemCtlPf* event counts the number of memory requests initiated by the memory controller prefetcher, and *MemCtlRd* counts the total number of memory read requests. These events are listed in Table I, and are measured using the hardware performance counters present on the AMD 10H architecture.

In Section V, we evaluate the performance of the hardware prefetchers by measuring their impact on the running time of five full-scale production applications.

1) Maximum stride: To understand the maximum streaming stride recognized by the two AMD prefetchers, we collected hardware counter events using configuration *MaxStride* shown in Table II, while enabling one prefetcher at a time. Figure 1 plots the prefetching effectiveness computed for each of the two hardware prefetchers as we varied the memory access stride between -8 and +8 cache lines. We also plot the number of L1 cache misses normalized to the total number of L2 requests originating from the data cache. As explained in Section III-A, the MC prefetcher brings prefetched data

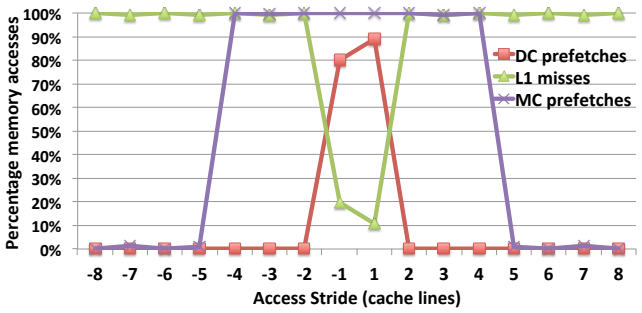


Fig. 1. Prefetch effectiveness as a function of access stride.

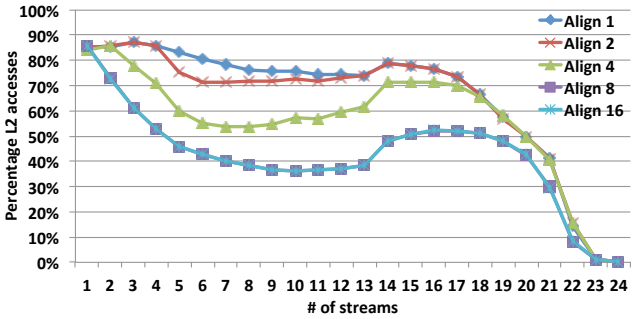


Fig. 2. DC prefetcher effectiveness as a function of stream count and page alignment.

into a buffer located at the memory controller level. Thus, a successful prefetch does not lower the observed number of L3 misses, but only their latencies.

The data clearly shows that the DC prefetcher recognizes only streams with a stride of one cache line, while the MC prefetcher recognizes streams with a stride of up to four cache lines. The two prefetchers recognize both forward and backward streaming accesses. We also notice that the DC prefetcher does not initiate all the data requests to L2 even for our simple, synthetic access pattern. Streams recognized by the DC prefetcher live only inside one page. A stream is discarded on a page boundary and a new stream must be recognized inside the new page. It takes three strided accesses to recognize an address stream. Thus, a number of accesses to each page are not prefetched.

2) Table size and Associativity: Because prefetch streams live only inside one memory page, and based on empirical observations from our initial experiments that suggested the occurrence of some type of resource conflict based on the page index of the streams, we believe that the DC prefetcher has at least one low associativity hardware structure that is indexed by the low bits of the page number. To verify these observations, we performed a set of experiments where we forced the streams to be separated by different powers of two numbers of pages. We used configurations *Align 1* to *Align 16* from Table II to understand the size of the hardware structures in the two prefetchers, and to determine if there are any associativity effects.

Figure 2 shows the DC prefetcher's effectiveness for different numbers of concurrent streams, from 1 to 24, and for different page alignments among streams. Each curve corresponds to the streams being separated by a multiple of the specified

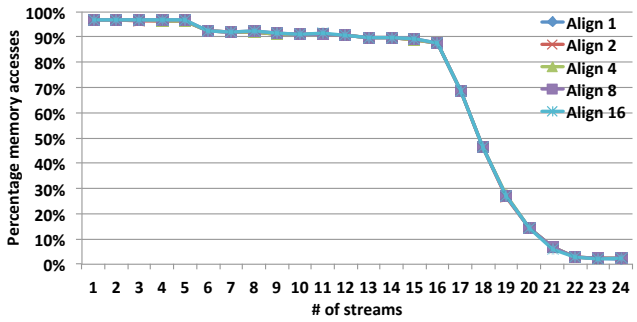


Fig. 3. MC prefetcher effectiveness as a function of stream count and page alignment.

number of pages, but not by a multiple of a larger power of two, see also the discussion at the start of Section III-B. We can make several observations based on these empirical results. Looking at the data points corresponding to `x` values of 1 to 8 streams for the different stream alignments, we observe a pattern that suggests the presence of a low associativity hardware structure.

Let the curve labeled *Align 1* be the baseline configuration. When stream locations are aligned by a multiple of 2 pages, the prefetcher performs as well as the baseline configuration up to a number of 4 streams. After that, its effectiveness decreases. When streams are aligned by 4 pages, the prefetcher performs at the baseline level up to 2 streams, while for 8- and 16-page aligned streams, the prefetcher's performance decreases for any number of streams larger than 1. These observations suggest the presence of a direct mapped, 8 entries structure in the DC prefetcher, which is indexed by the page number. However, the DC prefetcher continues to perform reasonably well up to 16-20 streams, which indicates that the 8-entries hardware structure is not the main limiting factor. Its direct-mapping, however, affects the prefetcher's effectiveness, especially for page alignments of 4-pages and larger powers of two.

Figure 3 presents the data collected for the MC prefetcher. Unlike the DC prefetcher, the MC prefetcher does not seem to be affected by the streams' page alignment. We should note, however, that our alignment is enforced on virtual addresses, and the MC prefetcher operates on a stream of physical addresses. Even so, the MC prefetcher seems to have a more consistent behavior, with only a slight decrease in effectiveness for more than five concurrent streams. The prefetcher's performance drops more sharply after 16 streams, suggesting the presence of a 16-entries hardware structure. However, the fact that its effectiveness drops somewhat gradually, may still indicate that a hardware structure is set-associative.

3) Parallel performance: While the DC prefetcher is private to each core, the MC prefetcher operates on the stream of physical addresses generated by all the cores. In this section, we analyze briefly the MC prefetcher's effectiveness as we increase the number of processes generating memory accesses. To evaluate the parallel effectiveness of the MC prefetcher, we loaded different numbers of concurrent instances of the micro-benchmark running configuration *Parallel* from Table II, from one to six concurrent instances.

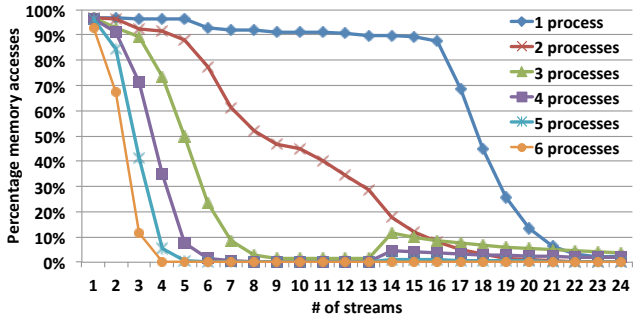


Fig. 4. Parallel MC prefetch effectiveness.

Figure 4 presents the fraction of memory accesses that have been prefetched as we varied the number of benchmark instances and the number of streams per process. The results show the prefetcher's effectiveness decreasing rapidly as we increase the number of processes. The prefetcher significantly underperforms relative to its single process performance. The MC prefetcher can handle well only 8-10 total streams across all processes in a multi-process configuration, with 10 streams performing well only in a two process configuration. These numbers reinforce the idea that the MC prefetcher has a set-associative hardware structure. We do not understand well how the indexing in this structure works, but the conflicts are exacerbated by increased process concurrency. Thus, at full core occupancy on the Istanbul micro-architecture, the MC prefetcher performs well only with a single stream per core. This limitation is likely to adversely affect the scaling of applications from one to six cores.

IV. Experimental Methodology

A. Application Preparation

As noted in the introduction, we ran our experiments on representative full-scale production applications from five important Exascale target areas: fusion energy (GTC), climate (CAM-HOMME), molecular dynamics (LAMMPS), materials science (NEK), and combustion (S3D). For each application we used sample-based profiling (HPCToolkit [2]) to identify five dominant loop nests. For brevity, in Section V, we refer to these loop nests using only the identifiers *loop* $1-5$ for each of these applications. We attempted to choose top-level loops close to the end of the call chain. No loops contain MPI library calls, eliminating potential impact of variable message latencies. We then 'calipered', i.e., marked the beginning and the end, the chosen loops with calls to the API of another tool (Memphis [19]), which collects performance counter data with very low overhead, allowing precise attribution of counted events to loops.

B. Hardware Experiments

On AMD hardware, bits in model-specific registers (MSRs) determine whether the hardware prefetching structures described in Section III-A are operational. We have implemented a user space tool that accesses the MSRs to enable or disable the hardware prefetch mechanisms on demand.

We enable and disable software prefetching through the use of PGI compiler (version 12.3 [20]) flags, creating two executables, one with prefetching instructions inserted and one without. PGI's '-fastsse' level of optimization, often used by application teams when running on high performance computing systems, automatically inserts prefetch instructions where it deems suitable. We use the '-Mnoprefetch' flag to disable prefetching. We have made no attempt to otherwise control the compiler's decisions about prefetching.

Of the eight possible combinations (MC on or off, DC on or off, and SW on or off), we chose to measure six:

- N: No prefetching, hardware or software, enabled.
- S: Only software prefetching enabled.
- M: Only memory controller prefetcher (MC) enabled.
- D: Only data cache prefetcher (DC) enabled.
- H: Both hardware prefetchers, MC and DC, enabled.
- HS: All prefetching mechanisms, hardware and software, enabled.

For each of the six combinations, we ran each application ten times, one for each event set described in Table I, once in serial mode and once in parallel (MPI). The parallel runs used all available cores in a single socket. We pinned processes to cores, in both serial and parallel runs, using the 'rankfile' mechanism provided by the OpenMPI [12] implementation.

Finally, we performed the full experiment on two AMD 10H platforms: a Shanghai implementation with four cores per socket, and an Istanbul implementation with six cores per socket.

C. Stream Simulator Experiments

To help us understand the observed prefetching performance on the full-scale production applications, we wrote a tool to abstractly understand the number of concurrent streams in an application [18]. The tool, written on top of PIN [16], works on unmodified and fully-optimized x86-64 binaries. In fact, for the software simulation runs of the full applications, we used the same executables as for the hardware measurements.

On each memory access, the simulator detects if the access is part of a stream or not. If the access is part of a stream, it also computes the number of concurrent streams active at that time. Optionally, memory accesses can be filtered by a configurable cache simulator. In this case, only cache misses are further classified as streams or not. For our full application results in Section V, we performed two simulation runs: 1) one simulation corresponding to the DC prefetcher detects streams with a stride of +/- 1 cache lines, and accesses are filtered by a 64KB, 2-way set-associative cache; 2) a second simulation corresponding to the MC prefetcher detects streams with a stride of up to four cache lines, and accesses are filtered by a 6MB, 48-way set-associative cache.

We stress tested the tool by analyzing memory access patterns generated by different synthetic micro-benchmarks, and the tool correctly identified streaming behavior in each case. Figure 5 shows a simple validation of the stream simulation algorithm for the micro-benchmark presented in Section III-B with up to 12 concurrent streams. The figure presents the

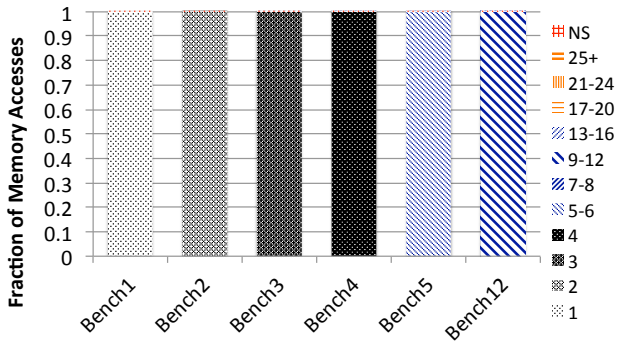


Fig. 5. Stream count simulation for micro-benchmark runs with different numbers of streams.

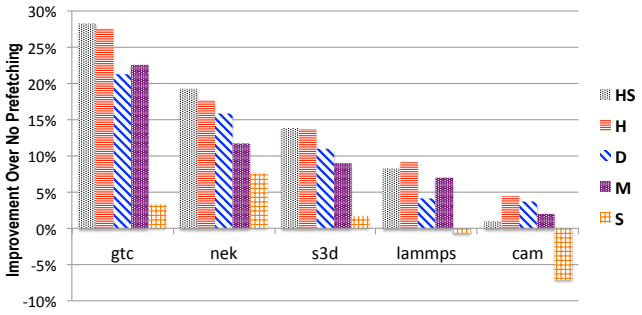


Fig. 6. Performance improvement over no prefetching (N) $((cycles - N\_cycles)/N\_cycles)$ for the following combinations of prefetching strategies: software only (S), memory-controller prefetcher only (M), data-cache prefetcher only (D), both hardware prefetchers (H), both hardware prefetches with software (HS).

distribution of stream counts detected by the tool for several runs of the micro-benchmark using different stream counts.

Each bar in the figure represents one execution of the micro-benchmark, using one, two, three, four, five, and twelve concurrent streams, respectively. The entries in the graph's legend show the different categories in which a memory access can be classified, representing how many parallel streams are active at that time. If a memory access is not part of any stream, it is classified instead as *NS* (Not a Stream). On the y axis we stack the normalized counts of memory accesses that fall into each of the categories listed in the legend. The stacks must sum up to 1 in each case.

The main loop of the micro-benchmark executes perfectly strided accesses. The simulation tool correctly detects this behavior and the actual number of streams in each benchmark execution. While these tests are very simple, they give us an opportunity to explain how the stream simulation data is presented in the Results section.

V. Experimental Results

A. Serial Results

Figures 6 and 7 collect results from serial runs. Figure 6 demonstrates the overall performance improvements of prefetching mechanisms, in isolation and working cooperatively, over no prefetching (N).

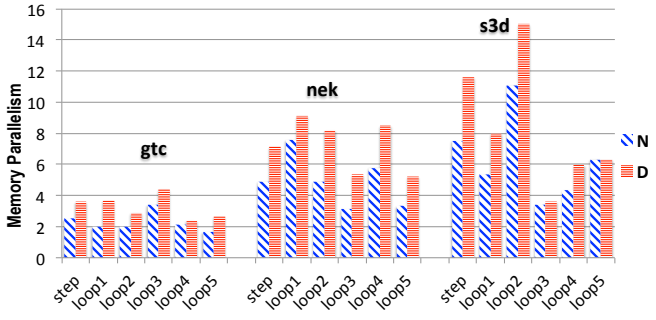


Fig. 8. Inherent memory parallelism (N) and increase due to DC prefetcher (D).

Hardware prefetching (H) always helps, improving performance by as much as 27% for *GTC*. Though in all applications other than *LAMMPS*, the DC prefetcher alone is responsible for a large fraction of the gain, the two hardware prefetchers are always more effective in tandem than alone, providing up to a 5% improvement over the best individual result (again in *GTC*). On the other hand, while software prefetching (S) alone helps *GTC*, *NEK*, and *S3D* by up to 5%, it degrades *LAMMPS* performance slightly, and *CAM* performance significantly (7%).

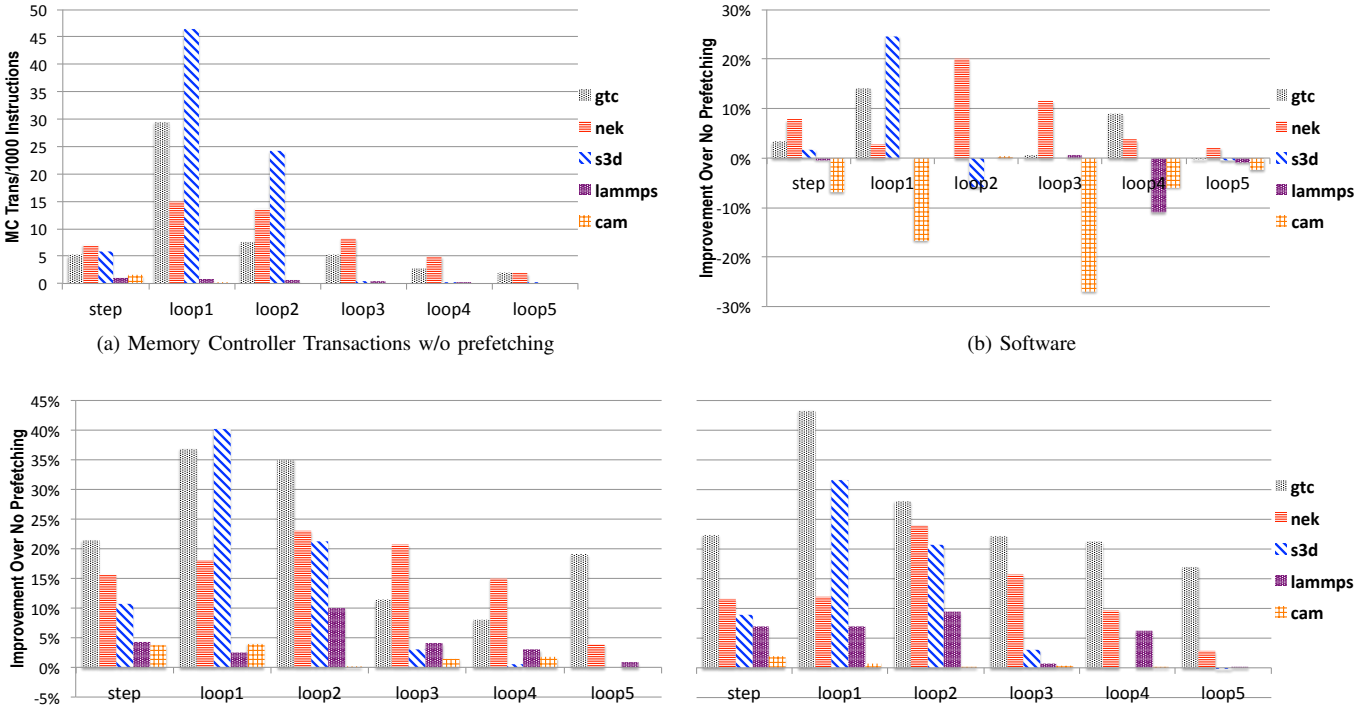
1) Memory Traffic: Figure 7a summarizes the memory requirements of the applications when prefetching is turned off, overall (in the 'step' bin) and by each representative loop nest, as measured in memory controller transactions per thousand instructions. Note that the loop nests are ordered high-to-low by memory requirement, and that the ordering is preserved in the remaining figures.

While the *GTC* and *NEK* loop nests demonstrate a fairly uniform and relatively high memory requirement, reflected in the overall 'step' numbers, *S3D*'s overall need, also high, is attributable to the extreme traffic generated by only two loop nests. In contrast, the working sets of *LAMMPS* and *CAM* appear to fit comfortably in cache, resulting in exceedingly low traffic to memory.

Figures 7c and 7d isolate the performance improvements of the DC and MC prefetchers respectively. Note the correlation between the loop nests with the highest memory requirements and those that see the greatest benefit from prefetching. The same result holds at the application step level.

Finally, Figure 7b isolates the impact of software prefetching. In most cases (*CAM* and *LAMMPS*), performance degradation corresponds to low memory traffic, likely due to replacement of important data close to the processor with unneeded data from memory. This conclusion is supported by significantly higher memory traffic for these loop nests when software prefetching is enabled (not shown). The one exception is the second loop nest in *S3D*, in which the compiler-generated prefetches clearly get in the way of heavy demand traffic.

2) Memory Parallelism: Figure 8 demonstrates the base memory parallelism (i.e., with no prefetching enabled) for the three memory-intensive applications, and the additional memory parallelism provided by the dcache prefetcher. We arrive at these numbers by dividing total DRAM accesses by



(a) Memory Controller Transactions w/o prefetching

(b) Software

(c) DC prefetcher

(d) MC prefetcher

Fig. 7. Performance improvement over no prefetching.

the results of counters that measure *non-overlapped* requests to DRAM.

We start by noting that the base memory-parallelism is quite high for *NEK* and *S3D*, over 4 and close to 8 respectively for a full step. Parallelism is lower for *GTC*, due to a large number of indirect array accesses that serialize many memory references. Nevertheless, the DC prefetcher substantially increases parallelism for all applications, by more than 50% overall. Again, in general the effect is greater for loops with higher memory requirements.

B. Parallel Results

Figure 9 presents results for parallel runs using all available cores on Shanghai (9a) and Istanbul platforms (9b). On Shanghai, while the DC prefetcher performance is quite stable between serial and parallel runs, the improvement due to the MC prefetcher alone for *GTC*, *NEK* and *S3D* declines markedly.

In all cases, the loss in productivity leads to a significant decrease in the performance boost due to hardware prefetching (H) from serial to parallel runs, accounting for a full 5% decrease for *GTC*. Based on analysis of loop level results, we find that the decrease in M productivity is due to a combination of contention for stream resources in the prefetching mechanism, and memory bandwidth constraints.

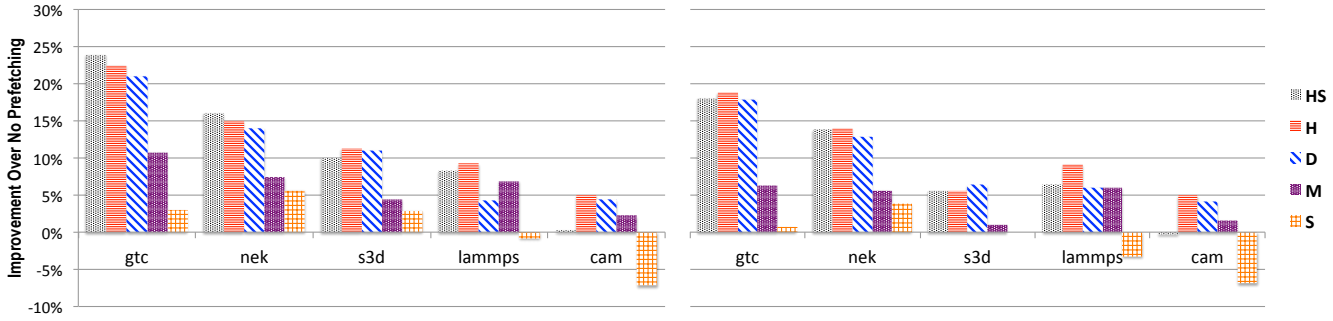
1) GTC: The hardware prefetchers produce significantly lower performance improvements during the parallel runs for three loop nests in GTC: *loop 1* (-18%), *loop 4* (-15%), and *loop 3* (-9%). In all cases, drops in MC performance gains substantially outweigh drops in DC prefetcher gains.

While *loop 1* has significant memory bandwidth requirements, with per-core memory bandwidth dropping from 3.2 GB/s in serial mode to 2 GB/s in parallel, the other two have more modest requirements and do not suffer a significant drop. However, both suffer from a marked drop in prefetch efficiency, as measured by the percentage of memory accesses due to prefetches: prefetch efficiency drops from 95% to 9% for *loop 4* , and from 98% to 21% for *loop 3*.

Interestingly, constructive interference from extra memory references due to the addition of DC prefetcher in H and HS runs slightly improves the MC prefetch efficiency for both *loop 4* and *loop 3*. However, for *loop 1*, the addition of DC prefetcher requests introduces *destructive* interference in the parallel case, reducing efficiency from 97% to 68%.

Stream simulation results, presented in Figure 10, explain both the benefit of MC in the serial case, and the decrease in its benefit in the parallel case. Figure 10a demonstrates that most DC misses in *loop 1* and *loop 4* are not streaming accesses, and the majority of references in *loops 2 and 3* require more than seven streams.

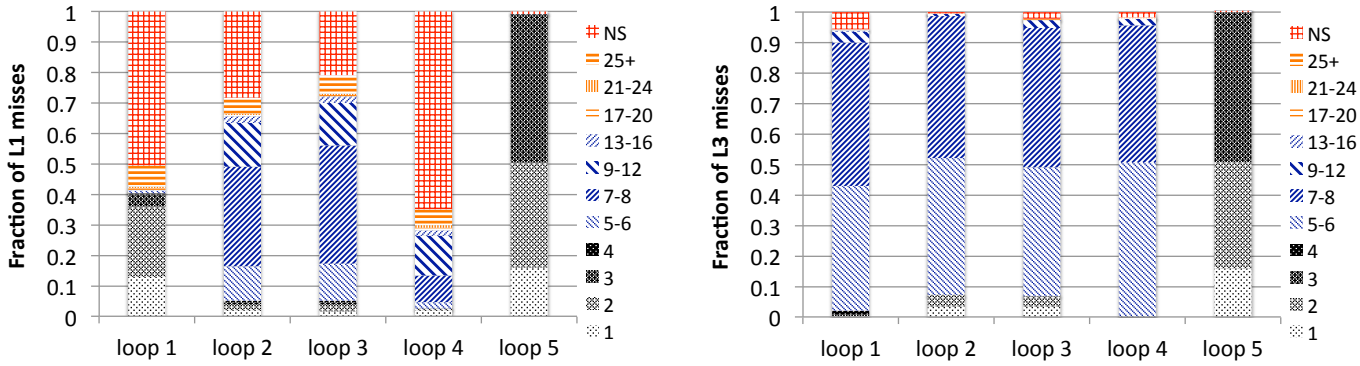
As noted earlier, *GTC* features a large number of indirect accesses. The substantial reduction in non-streaming accesses observed in Figure 10b demonstrates that the L3 cache acts as a filter, removing irregular reference patterns that confuse the DC prefetcher, and leaving a small number of regular streams for the MC prefetcher. Furthermore, the simulation tool reveals that *loop 1* features a stride-4 reference pattern, due to the memory layout of the inner dimension of two dominant array variables, which cannot be detected by the DC prefetcher. Figure 10b also shows that the first four loop nests in GTC



(a) Shanghai parallel (4 cores)

(b) Istanbul parallel (6 cores)

Fig. 9. Performance improvement over no prefetching (N) for parallel (MPI) runs of the applications.



(a) L1 Miss Stream Coverage

(b) L3 Miss Stream Coverage

Fig. 10. GTC serial stream simulation results. Each bar on the x axis corresponds to one loop nest. On the y axis we have the percentage of cache misses that fall into each category listed in the graph's legend, depending on the number of concurrent streams active at that time.

require 5 to 8 streams per process at the memory controller level, and Figure 4 showed us that the MC prefetcher becomes much less effective in a multi-process configuration at higher levels of streaming concurrency.

2) S3D and NEK: Two loop nests in *S3D* and one in *NEK* exhibit similar behavior with respect to MC prefetcher performance. Figure 11 presents stream simulation results for *S3D*. In contrast with *GTC*, Figure 11a shows that the streams in both *S3D* loops should easily be covered by the DC. However, since the L3 miss rate is extremely high for these loops, the MC prefetcher sees very nearly the same streams. Though the number of streams at the MC is quite low, and therefore should not necessarily be subject to contention from other MPI processes, we still see a substantial reduction in prefetch efficiency in the hardware results, particularly for the second loop nest, likely due to associativity conflicts.

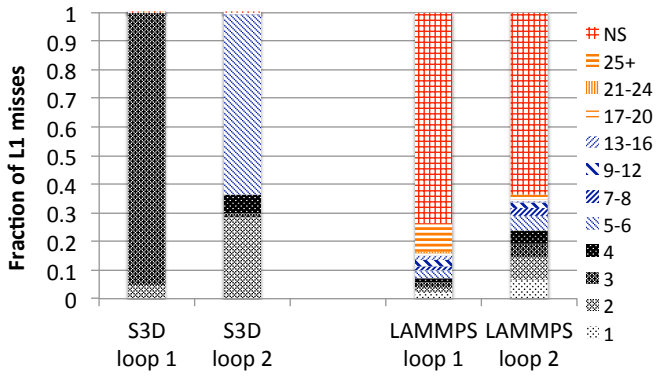
3) LAMMPS: Finally, *LAMMPS* is unique in that the MC prefetcher improves performance of the dominant loop nest (*loop 1*), and the contribution is not diminished during a parallel run. The stream simulation results in Figure 11 again explain the reason for this behavior. Based on the Figure 11a results, the *loop 1* accesses that the DC prefetcher sees are for the most part not streamable. However, the filtered accesses that the MC sees easily fit into a single stream, in other words the accesses come from a single program array. The program-level explanation for this behavior is that the loop nest computes interactions between atoms and while access to the atoms is highly irregular, access to the *neighbor* list is regular. While accesses to the atom data exhibits reuse in L3, the interaction list is linearly traversed with no temporal reuse inside the loop.

VI. Conclusion

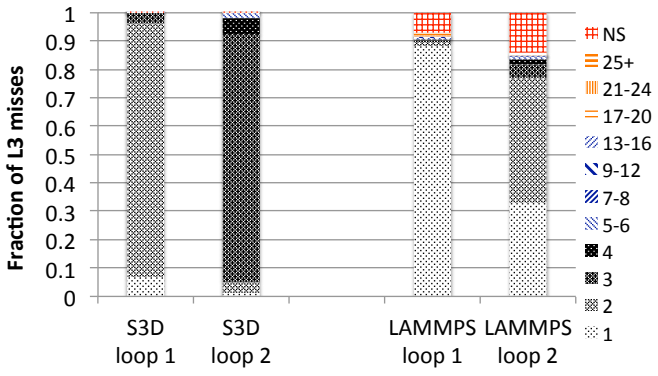
In order to better understand the impact of data prefetching on scientific application performance, this paper has introduced two techniques, one *micro-architecture*-centric and the other *application*-centric, to analyze the prefetching performance of five important Exascale target applications.

We have found that despite a great diversity in prefetching effectiveness, there is a strong correlation between regions where prefetching is most needed (due to high levels of memory traffic) and where it is most effective. Further, we found that while hardware prefetching always improved application performance in the applications we studied, compiler-inserted software prefetches degraded performance for two applications with extremely low memory requirements.

We also found that a hardware prefetcher at later levels of the memory hierarchy can boost *serial* performance significantly, but contention for the shared resource can substantially reduce effectiveness in a parallel context. Finally we found that prefetching substantially increases already high levels of



(a) L1 Miss Stream Coverage



(b) L3 Miss Stream Coverage

Fig. 11. S3D and LAMMPS serial stream simulation results.

memory parallelism, *even in applications that prominently feature irregular data accesses*.

REFERENCES

[1] Optimizing embedded system performance–impact of data prefetching on a medical imaging application, 2006. 2

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. 5

[3] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*, 2010. 2

[4] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 10h and 12h Processors*, 2011. 1

[5] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. M. A. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande. A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage. *IEEE Journal of Solid-State Circuits*, 44(1):217–227, 2009. 1

[6] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 40–52, New York, NY, USA, 1991. ACM. 2

[7] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 223–232, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 1

[8] Y. Chen and X. Wang. Compact modeling and corner analysis of spintronic memristor. In *IEEE/ACM International Symposium on Nanoscale Architectures 2009 (Nanoarch)*, pages 7–12, 2009. 1

[9] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-aware shared resource management for multi-core systems. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 141–152, New York, NY, USA, 2011. ACM. 1

[10] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 316–326, New York, NY, USA, 2009. ACM. 1

[11] S. A. et al. The opportunities and challenges of exascale computing. Technical report, U.S. Department of Energy, 2010. 1

[12] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 5

[13] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *Proc. International Electron Device Meeting Tech. Dig.*, pages 459 – 462, 2005. 1

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 364–373, New York, NY, USA, 1990. ACM. 2

[15] F. Liu and Y. Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 39(1):37–48, June 2011. 2

[16] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. 5

[17] C.-K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Trans. Comput.*, 48(2):134–141, Feb. 1999. 2

[18] G. Marin, C. McCurdy, and J. S. Vetter. Diagnosis and optimization of application prefetching performance. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 303–312, New York, NY, USA, 2013. ACM. 2, 5

[19] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Proc. of the 2010 IEEE Intl. Symp. on Performance Analysis of Systems Software*, pages 87–96, March 2010. 5

[20] Portland Group International, Inc. *PGI Compiler User's Guide*, 2012. 5

[21] T. R. Puzak, A. Hartstein, P. G. Emma, and V. Srinivasan. When prefetching improves/degrades performance. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 342–352, New York, NY, USA, 2005. ACM. 2

[22] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data prefetching on the hp pa-8000. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 264–273, New York, NY, USA, 1997. ACM. 2

[23] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 63–74, Washington, DC, USA, 2007. IEEE Computer Society. 1