Diagnosis and Optimization of Application Prefetching Performance

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

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

Jeffrey S. Vetter
Future Technologies
Oak Ridge National Laboratory
vetter@ornl.gov

ABSTRACT

Hardware prefetchers are effective at recognizing streaming memory access patterns and at moving data closer to the processing units to hide memory latency. However, hardware prefetchers can track only a limited number of data streams due to finite hardware resources. In this paper, we introduce the term *streaming concurrency* to characterize the number of parallel, logical data streams in an application. We present a simulation algorithm for understanding the streaming concurrency at any point in an application, and we show that this metric is a good predictor of the number of memory requests initiated by streaming prefetchers. Next, we try to understand the causes behind poor prefetching performance. We identified four prefetch unfriendly conditions and we show how to classify an application's memory references based on these conditions. We evaluated our analysis using the SPEC CPU2006 benchmark suite. We selected two benchmarks with unfavorable access patterns and transformed them to improve their prefetching effectiveness. Results show that making applications more prefetcher friendly can yield meaningful performance gains.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Performance, Algorithms, Experimentation

Keywords

Stream prefetching, Performance modeling, Diagnosis

1. INTRODUCTION

The gap between CPU and memory speeds has constantly increased over the last few decades. Efforts have been made on both the hardware and software sides to improve system throughput by hiding part of the memory latency. One of the techniques most effective at hiding memory latency is to increase memory parallelism. Memory parallelism, or memory concurrency, can be increased by different means. One approach is to increase the number of processing cores connected to the memory system, another approach is to increase hardware threading concurrency through the use of simultaneous multithreading techniques, and finally, we can increase memory concurrency inside each thread.

Mandal et al. [11] empirically evaluate the level of memory parallelism per core, socket and node, needed to saturate the memory system of several current micro-architectures. Increasing memory parallelism beyond the capacity of the memory system does not bring direct performance benefits. However, increasing the level of memory concurrency inside each thread, ensures that an application runs efficiently in both low-threading and high-threading environments.

Data prefetching, that is, identifying data that will be needed in the near future and moving it closer to the processing units so that it is available when required by a running application, is an effective optimization both for increasing memory parallelism inside a thread and for overlapping memory latency with computation. Prefetchers based on Jouppi's stream buffers [8] are nowadays commonly found in modern micro-processors. These hardware structures recognize streaming memory access patterns and fetch the next elements of the streams in advance to hide memory latency. However, applications benefit differently from hardware prefetching, because they either do not traverse memory in a streaming pattern, or they exceed the resources provisioned for the hardware prefetchers.

In this paper, we describe techniques for modeling the streaming behavior of applications. We use these models to explain why applications benefit differently from hardware prefetching and to identify the causes behind poor prefetching performance. We use the SPEC CPU2006 benchmark suite [7] to evaluate our analysis. We selected two benchmarks with unfavorable access patterns, 436.CactusADM and 470.LBM, and changed them to improve their prefetching coverage. Our results show that tuning for the hardware prefetchers can yield meaningful performance benefits.

The rest of the paper is organized as follows. Section 2 provides a brief overview of related work. Section 3 presents background information on the AMD 10H hardware prefetchers. Section 4 describes a simulation algorithm for computing the *streaming concurrency* of applications, and evaluates its predictive power. Section 5 refines the simulation results using static analysis to identify the causes of poor prefetching performance. Section 6 presents the results of our analysis applied to the SPEC CPU2006 benchmark suite and describes two tuning case studies. Section 7 summarizes the findings of this study and concludes the paper.

2. RELATED WORK

Prefetching comes in two main flavors, software prefetching and hardware prefetching. Both techniques have been extensively studied in the past. Literature on software prefetching focuses primarily on algorithms for understanding where compilers should automatically insert prefetch instructions [4, 15, 10]. 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 [5, 17, 6].

Palacharla and Kessler [14] describe an algorithm for detecting streams as part of a stream buffer simulator. We improved on this algorithm and extended it to recognize streams with non-unit strides, and to compute the number of live data streams at any point in an application. We use streaming concurrency results in combination with static program analysis to understand which memory access patterns of an application are not effectively prefetched.

Williams et al. [19] have recognized the importance of writing code that is friendly to the hardware prefetchers. However, their optimizations are based on manual inspection of the code. We describe an analysis technique and tool implementation that abstractly understands streaming behavior in applications and provides insight into prefetch unfriendly memory access patterns. To our knowledge, this is the first description of such a tool.

3. BACKGROUND

The AMD 10H micro-architecture, represented by the Barcelona, Shanghai and Istanbul chips, has two streaming hardware prefetchers. The first prefetcher is associated with the data cache level and is replicated across all the cores of a microprocessor. We call it the DC prefetcher. The DC prefetcher monitors memory accesses that miss in the L1 cache. It tries to detect streaming data access patterns, and speculatively fetches the memory addresses predicted to be accessed in the near future. The second prefetcher is associated with the memory controller. Therefore, we call it the MC prefetcher. The MC prefetcher operates on the stream of memory addresses generated by misses in the last level of cache from all the cores connected to that memory controller. Because it operates on memory accesses that are filtered by the L3 cache, typically, it can discover a different set of data streams than the DC prefetcher, even when only a single core is active and generating addresses. The MC prefetcher fetches data into a separate prefetch buffer to avoid conflicts with the data loaded into the CPU caches.

Based on micro-benchmark testing, we found that the DC prefetcher recognizes streams with a maximum stride of one cache line, while the MC prefetcher understands streams with a stride of up to four cache lines. Both prefetchers recognize streams with either positive or negative strides. Hardware prefetching structures have limitations similar to other caching mechanisms. They keep track of streaming accesses that reference consecutive or nearby lines. However, successive accesses to nearby lines must exhibit a certain level of temporal locality, because the hardware structures

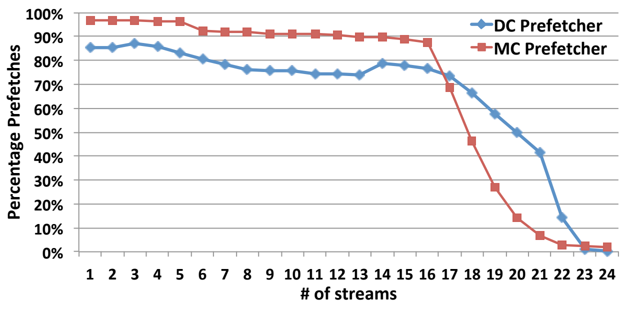


Figure 1: Prefetch requests as a function of stream count.

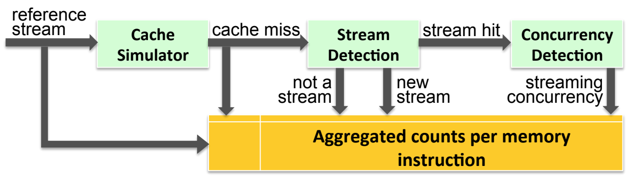


Figure 2: Diagram of the stream simulation tool.

used by the prefetchers have finite capacities. Again, we used micro-benchmarks to empirically understand the maximum number of data streams that can be handled by each AMD 10H hardware prefetcher.

We wrote a micro-benchmark that streams over a block of memory using a configurable number of concurrent streams. Figure 1 presents the fraction of memory requests that have been initiated by each of the hardware prefetchers as we varied the stream count. We measured these numbers using the hardware performance counters available on the AMD 10H architecture. The data in Figure 1 shows that the MC prefetcher is effective for up to 16 concurrent streams. This suggests the presence of a 16-entries hardware structure. The effectiveness of the DC prefetcher seems less stable. Overall, it stays relatively high up to a concurrency level of 16, and then decreases slowly up to a concurrency of 21 streams. However, its overall effectiveness is lower than that of the MC prefetcher for up to 16 streams.

4. STREAMING CONCURRENCY

Applications benefit from hardware prefetching in different degrees. To explain these differences, we developed a tool for identifying streaming behavior and for building a streaming concurrency model of an application. Our thesis is that the streaming concurrency model of an application can explain its observed prefetching performance.

We implemented a PIN [9] based tool to detect streaming behavior in optimized x86 binaries. Figure 2 shows the main components of our simulation tool. First, memory references are processed by a cache simulator, because the streaming hardware prefetchers on current micro-architectures are placed behind one or more levels of cache, with the cache acting as a filter on the address stream. Our cache simulator uses a true least recently used (LRU) replacement policy, but its capacity, block size and associativity are fully customizable otherwise. Accesses that miss in cache are further processed by a stream detection module, which is explained in the next section.

4.1 Stream Detection

Our stream detection algorithm is based on the stream buffer simulation algorithm described in [14]. We added extensions for detecting streaming behavior with non-unit strides. Because this algorithm is implemented in software, we could use a more general solution that does not require either knowing the PC of the instructions that perform memory accesses [1], or partitioning the address space and limiting stride detection within each partition [14].

Our stream detection algorithm uses two primary data structures, a *Stream Table* that stores information about streams already recognized, and a *History Buffer* that keeps track of memory locations recently accessed. The latter data structure is used to recognize new streams. A new stream is formed when two strided accesses, with the same stride, are detected. That is, the program must access locations i, $i + s$ and $i + 2s$ to form a new stream with stride s, whose next access is expected to be at location $i + 3s$.

The history table is a hybrid ring buffer / balanced search tree data structure. The ring buffer organization maintains a FIFO ordering of the entries. The balanced search tree organization is used to efficiently locate entries corresponding to nearby locations in memory for a history table that may be large in size. The history table entries are C structures with the following fields: *loc* - stores the memory location associated with an entry and is also the sorting key for the balanced search tree; *pstrides* and *nstrides* are two bit sets that keep track of positive and negative strides, respectively, of potential streams that include this memory location.

The stream table is also a hybrid data structure, organized both as a doubly linked list to maintain an LRU ordering between entries, and as a hash table to enable rapid detection of a stream hit. A stream table entry has the following fields: *next_loc* stores the next memory location expected to be accessed by a stream and is also the key used for hashing; *stride* stores the stride associated with a stream and is used to advance the stream on a stream hit; *prev* and *next* are pointers to other stream table entries, which are used to implement the doubly linked list view of the table.

Algorithm 1 shows the outline of the stream detection algorithm. On a memory access, we first check if the accessed memory location is predicted by any of the streams already stored in the *Stream Table*. If the location has been predicted by one of the streams, we call the access a *stream hit*. In this case, we only have to advance the stream, which consists of deleting its current entry from the hash table, updating the expected memory location field using the stride information, rehashing the entry based on the new value of the *next_loc* field, and updating the *next* and *prev* pointers to move the accessed stream into the most recently used (MRU) position. In addition, on a stream hit, we compute the instantaneous streaming concurrency value (see §4.2).

In case of a stream miss, we must check if the new memory access appears to be in a streaming pattern with other recently accessed locations stored in the history buffer (lines 10-31), allocate a new stream in the stream table if a stream is detected (lines 33-37), and finally, add a record of the newly accessed memory location to the history buffer (lines 38-44).

Algorithm 1 Stream detection algorithm.

```
 1: HistoryEntry histBuffer[H]
 2: Tree<HistoryEntry*> histTree
 3: function StreamDetect($mloc, sstride$)
 4:     # $mloc$ - accessed memory location
 5:     # $sstride$ - stream stride (output parameter)
 6:     $stream \leftarrow FindStream(mloc, sstride)$
 7:     if $stream \neq 0$ then
 8:         return $AdvanceStream(stream)$
 9:     else  # stream miss, add to the history table
10:         $uElem \leftarrow histTree.upper\_bound(mloc + M)$
11:         $lElem \leftarrow histTree.lower\_bound(mloc - M)$
12:         $pstrides \leftarrow 0$
13:         $nstrides \leftarrow 0$
14:         $min\_stride \leftarrow MAX\_INT$
15:         $min\_elem \leftarrow 0$
16:         for all $lElem \leq elem < uElem$ do
17:             $s \leftarrow mloc - elem.loc$
18:             if $s > 0$ then
19:                 $pstrides.Set(s)$
20:                 if $elem.pstrides.IsSet(s)$ && $s < abs(min\_stride)$ then
21:                     $min\_stride \leftarrow s$
22:                     $min\_elem \leftarrow elem$
23:                 end if
24:             else if $s < 0$ then
25:                 $nstrides.Set(-s)$
26:                 if $elem.nstrides.IsSet(-s)$ && $-s < abs(min\_stride)$ then
27:                     $min\_stride \leftarrow s$
28:                     $min\_elem \leftarrow elem$
29:                 end if
30:             end if
31:         end for
32:         $status \leftarrow NO\_STREAM$
33:         if $min\_elem \neq 0$ then
34:             $status \leftarrow NEW\_STREAM$
35:             $sstride \leftarrow min\_stride$
36:             $AllocateStream(mloc, sstride)$
37:         end if
38:         if $histBuffer.IsFull()$ then
39:             $histTree.Erase(histBuffer[lastIdx])$
40:             $lastIdx \leftarrow (lastIdx + 1) \mod H$
41:         end if
42:         $histBuffer[top] \leftarrow \{mloc, pstrides, nstrides\}$
43:         $histTree.Insert(histBuffer[top])$
44:         $top \leftarrow (top + 1) \mod H$
45:         return $status$
46:     end if
47: end function
```

4.2 Concurrency Detection

On a stream hit, we want to understand how many other data streams are active at the same time. We call this metric the *streaming concurrency* of an application. Previous stream buffer simulations matched an existing or proposed hardware implementation. They were using a predefined number of buffer streams, and were only interested in finding how many accesses have been stream hits (prefetchable) or stream misses. While we also use stream tables and history buffers of predetermined sizes during simulation, these sizes can be large, and have no relationship to any existing micro-architecture. We use simulation to build a model of an application's memory access patterns.

On each stream hit, we compute a streaming concurrency value for that memory access. We define the concurrency

Table 1: Simulation parameters.

Param	Description	DCsim	MCsim
C	Cache capacity (in KB)	64	6144
L	Cache line size (in bytes)	64	64
A	Cache associativity	2	48
M	Max stream stride(lines)	1	4
H	History buffer size	256	256
T	Stream table size	128	128

of a stream hit as the number of streams that have been advanced more recently than the previous hit on the current stream. This metric captures how many other streams are advancing concurrently with the current stream, which corresponds to the number of data arrays that are being streamed over by an application. This metric is also equivalent to computing the LRU stack distance [13, 2] for streams.

We compute the LRU stack distance using a balanced binary tree [3], which provides an $O(log(T))$ execution time per access, where T is the size of the tree. Our implementation, based on [12], uses a splay tree [16], and because we use a fixed size stream table during the simulation, the balanced tree size is upper bounded by the table size.

We aggregate application profiles at the memory instruction level. We process this profile information in a post-mortem step to compute different statistics at instruction, loop and routine level.

4.3 Simulation Results

We compare our histograms of streaming concurrency against hardware counter measurements of prefetching requests initiated by the two AMD 10H streaming hardware prefetchers. Our motivation for this work is to explain observed differences in prefetching performance and to identify opportunities for tuning in scientific applications. For this evaluation, we use the full SPEC CPU2006 benchmark suite because it provides a broad set of applications and it eliminates the temptation of cherry picking a small number of HPC applications that reflect better on our analysis.

We use the `train` input set instead of the `ref` input set for our evaluation because despite a careful implementation, simulating each memory access causes ≈`60x` execution slowdown. In addition, we believe that performance diagnosis based on performance modeling does not need to be done at scale to be useful. Our models capture features of an application, such as streaming behavior, that are true at different scales. Our tuning results in §6 include data for `ref` inputs.

The stream detection algorithm abstractly identifies the number of concurrent streams in an application. This measure is a machine independent application characteristic. However, on most micro-architectures, including the AMD 10H architecture, stream buffers are placed after one or more levels of cache. Caches act as filters on the stream of memory addresses. Therefore, we configured our simulation tool with an architecture dependent cache simulator in the first stage, as shown in Figure 2. We used a dual-socket AMD Istanbul based machine, with two 6-core processors running at 2.6GHz and 12GB of main memory for both our simulations and our hardware counter measurement results.

We performed two simulation runs, using the configurations shown in Table 1: 1) one simulation targeting the DC prefetcher uses a 64KB, 2-way set-associative cache in the first stage and detects streams with a stride of +/- one cache line; 2) a second simulation targeting 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.

Table 2: SPEC benchmark names.

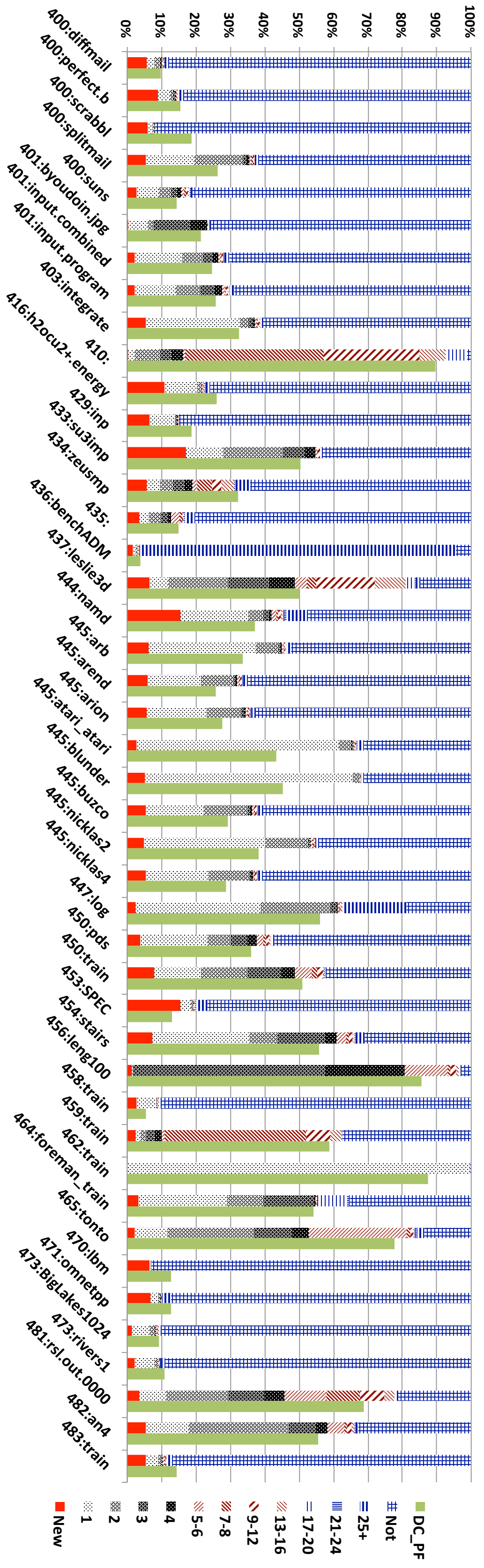
Index	Name	Index	Name
400	perlbench	401	bzip2
403	gcc	410	bwaves
416	gamess	429	mcf
433	milc	434	zeusmp
435	gromacs	436	cactusADM
437	leslie3d	444	namd
445	gobmk	447	dealII
450	soplex	453	povray
454	calculix	456	hmmer
458	sjeng	459	GemsFDTD
462	libquantum	464	h264ref
465	tonto	470	lbm
471	omnetpp	473	astar
481	wrf	482	sphinx3
483	xalancbmk		

Figure 3a shows the streaming concurrency results side by side with the hardware counter measurements of prefetch requests initiated by the DC prefetcher, for all the benchmarks and all the `train` inputs. On the `x` axis we show the benchmark index followed by a colon and the name of the input file, as it was captured by our automated script. For some benchmarks, no input file could be detected, so that information is missing. For brevity, we did not include the benchmark names in the figure labels. Table 2 shows the benchmark names associated with each benchmark index.

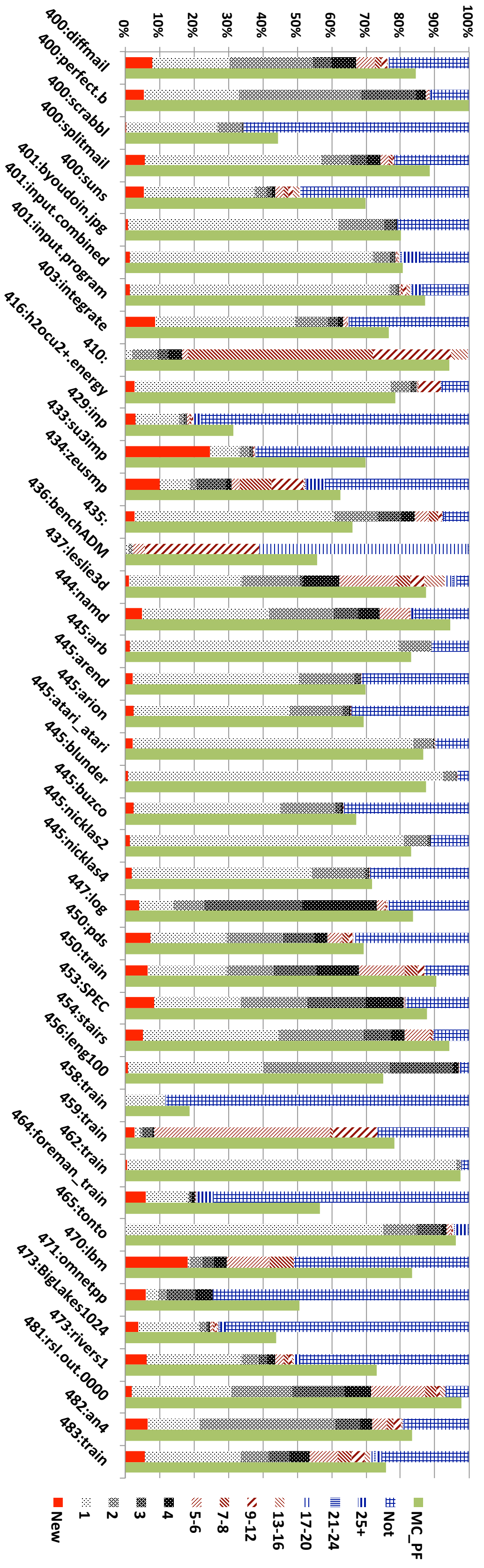
For each benchmark, we show a stacked histogram of the streaming simulation results aggregated at the entire program level. The bin values are normalized by the number of cache misses observed for each benchmark. Only cache misses are processed by the stream detection simulation, and thus, the bins of the histograms stack up to 1. The bins labeled as `New` reflect the fraction of cache misses that resulted in new streams being created. The streaming concurrency values, measured only for stream hits, have been aggregated into 11 distinct bins, with increasingly coarser resolutions. Finally, the `Not` bins show the fraction of non-streaming accesses observed for each benchmark. The `DC_PF` metric, measured using hardware counters, shows the fraction of data accesses to the L2 cache initiated by the DC prefetcher.

The bins are color coded for easier inspection, and use different hash patterns to be legible in black and white. Thus, low streaming concurrency bins, 1 to 4 streams, use dot patterns and are colored in black. Medium concurrency bins, 5 to 16 streams, use diagonal hash lines and are colored in red. Finally, high concurrency bins, 17 streams and higher, use horizontal or vertical hash lines and are colored in blue. Non-streaming accesses are plotted with a grid hash, and are also colored in blue. We specifically decided to make these distinctions between the different concurrency levels.

When a new stream is detected, the hardware prefetcher will attempt to fetch the next location predicted to be part of the stream. Similarly, when a stream is advanced on a stream hit, the hardware prefetcher will fetch the next location predicted by the stream. Based on the prefetching effectiveness profile shown in Figure 1, we expect that most accesses with a streaming concurrency of up to 16 will hit



(a) DC prefetcher



(b) MC prefetcher

Figure 3: Streaming concurrency and prefetching measurements for the two AMD 10H hardware prefetchers.

in the hardware prefetcher's own hardware structures, and thus result in a new prefetch request to the L2 cache. The differences in color and filling patterns between streaming concurrencies below 16 and those of 17 and higher, make it easier to compare the simulation and measurement results.

We opted to present a graphical, qualitative comparison of these two metrics, because first, the streaming concurrency histograms include more information than a single quantitative prediction of prefetch requests computed by summing up the `New` bin and the bins with a concurrency of up to 16, and second, we want to show that the streaming concurrency metric provides a good explanation of prefetching effectiveness, even as we do not fully understand the precise implementation of the AMD 10H hardware prefetchers. As can be seen in Figure 1, the DC prefetcher is not 100% effective even when we generate clean streaming accesses at low concurrency levels, using a synthetic micro-benchmark.

Overall, we believe the results in Figure 3a explain well the measured prefetch counts. Across the entire suite, almost half of all DC cache misses are not streaming accesses. However, for most benchmarks, memory accesses that have streaming behavior are characterized by a low or medium concurrency level, which means that they can be effectively prefetched by the current hardware prefetchers. Only benchmark `436.cactusADM`, and to a lesser extent benchmarks `447.dealII` and `464.h264ref`, have a significant fraction of streaming accesses characterized by high concurrency. There are also three outliers for which the measured number of prefetch requests is lower than what we would expect based on the concurrency histograms: benchmark `437.leslie3d`, and inputs `atari_atari` and `blunder` of benchmark `445.gobmk`. We note that in these cases, the number of accesses to the L2 cache[1], measured with hardware performance counters, was 25% to 40% higher than the miss counts produced by our L1 cache simulator. Benchmark `gobmk` is known to have very poor branch predictability, which likely leads to many wrong-path loads, not visible to our PIN based simulation.

Often, prefetchers may initiate unnecessary prefetch requests, on one hand due to inaccurate predictions, but also due to superfluous requests at the end of streams when applications stop accessing data along those same patterns. The number of newly created streams computed by our simulation is a good indicator of how many superfluous requests are generated at the end of streams, because all the streams that are created will stop advancing eventually. The number of unnecessary prefetch requests generated at the end of streams depends also on how many lines are fetched in advance by each prefetcher. In the case of the DC prefetcher, this number seems to be small.

Similarly, Figure 3b compares the simulated streaming concurrency and the number of prefetch requests observed for the MC prefetcher. The `MC_PF` metric represents the fraction of memory requests initiated by the MC prefetcher, measured using hardware counters. Most of the discussion for the DC prefetcher applies to the MC prefetcher results as well. However, evaluating the behavior of the MC prefetcher is complicated by additional factors. Our simulation analyzes the stream of memory accesses generated by the application and filtered by a cache with the characteristics of the L3 cache on the target machine. However, in practice, the DC prefetcher will generate additional memory requests that may be filtered or not by the intervening levels of cache. Also, the three levels of cache of the AMD 10H architecture are not in an inclusive relationship. Simulating only the last cache level does not capture precisely the number of memory accesses that are processed by the MC stream buffers.

[1]This is the denominator used for computing the DC_PF metric.

The results in Figure 3b show once again that the streaming concurrency metric explains well the measured MC prefetch counts. Benchmarks `433.milc` and `470.lbm` stand out with a high rate of stream creation. The high rate of stream creation results in a high number of superfluous prefetch requests at the end of the streams, which causes the measured prefetch counts to be higher than what we would expect. These results suggest that the MC prefetcher fetches multiple lines in advance, which is understandable considering the higher memory latency that it must hide.

5. PERFORMANCE INSIGHT

The simulation algorithm introduced in the previous section, abstractly identifies streaming behavior in applications. We performed simulations with a larger stream table size, to pinpoint cases when an application exhibits streaming behavior, but the number of streams exceeds the size of the hardware structures. However, even the data structures that we use in our simulations are finite in size, for performance reasons. Accesses with streaming concurrency levels higher than the size of the *Stream Table* are labeled instead as having non-streaming behavior. In this section, we refine the simulation results using static analysis. Besides improving the accuracy of the simulation results, we also want to identify opportunities for improving prefetching performance.

Our models assume that accesses that create new streams and streaming accesses with concurrency of up to 16 are prefetchable. We focus our attention on the other types of accesses. We base our analysis on the following observations:

1) Streaming accesses are strided accesses with a stride less than or equal to the maximum recognized stride, M. We generally think of streaming accesses as being generated by the loop immediately enclosing a memory instruction. However, consider the case of a two dimensional array laid out in row major order, as in C, and a two level loop nest where the inner loop iterates over the rows, and the outer loop iterates over the array's columns. Every iteration of the inner loop touches different rows, which are far apart in memory. Therefore, the inner loop does not create streaming accesses. However, on each iteration of the outer loop we access consecutive elements of each row. Thus, the outer loop advances a separate stream for each row of the array. We know that such an access pattern is sub-optimal due to long spatial reuse. However, it also creates a situation where, potentially, a very large number of streams are accessed in a round-robin fashion. We state that out of all the loops enclosing an instruction, the loop that produces the smallest access stride is the loop that creates streaming behavior.

2) Non-streaming accesses correspond either to non-strided memory instructions, also known as irregular accesses, or they correspond to strided accesses whose strides are larger than the maximum recognized stride.

Recall that our simulation tool uses PIN to understand streaming behavior in unmodified, optimized x86 binaries and that we collect streaming concurrency profiles at the instruction level. We use static analysis on the same application binaries to distinguish between the scenarios described above. We perform data flow analysis on a routine's control flow graph (CFG) to symbolically understand memory access patterns in applications. For each memory instruction, we build symbolic formulas that describe how the memory location accessed by that instruction changes from one iteration to the next of a particular loop. We compute one such formula for every loop level that encloses that instruction. These formulas describe the strides with which a memory instruction traverses over memory, with respect to each loop enclosing it. The data flow analysis also detects cases when the computed stride is not consistent across iterations, or when a memory access is indirect with respect to a loop. We classify these strides as *irregular*.

We use the derived stride formulas to further refine the simulation results for each memory instruction. We look at an instruction's stride formulas in order, from the innermost to the outermost loop, until we find an irregular stride, or until we exhaust all the loop levels. We want to find the smallest non-zero stride associated with an instruction and record the loop level that creates it. The smallest stride corresponds to the loop level that has the potential to stream over the associated source code data structure. We ignore strides that are equal to zero, because they correspond to loops whose index variables are not used to reference the source data structure. Repeated accesses to the same location do not create streaming behavior.

This first step of the analysis yields the following scenarios: 1) At least one loop creates a constant[2] non-zero stride. We record the loop that produces the smallest non-zero stride. 2) We find only symbolic strides that depend on run-time values. 3) We find an irregular stride before finding any non-zero stride. 4) We find no non-zero strides. Next, we look more closely at each of these scenarios.

1) If we find a smallest constant non-zero stride, S, we check if S is larger than the maximum recognized stride, M. If it is, then we classify all the non-streaming accesses produced by that instruction as being produced by an access pattern with a large stride. If $S \leq M$, this instruction has streaming behavior. If the simulation determined that this instruction produced any non-streaming accesses, it must be because the streaming concurrency level was higher than the size of the *Stream Table* data structure used in the simulation. We further check the position of the loop producing the smallest stride, S. If it is the loop immediately enclosing the instruction, we say that any non-streaming accesses are caused by a high inner concurrency level. Otherwise, we say that high outer concurrency causes non-streaming accesses.

2) If we find only symbolic strides, we can assume that the stride is too large to produce streaming behavior. However, for now, we classify such accesses as having a symbolic stride, to understand their rate of occurrence in real applications.

3) If the stride is irregular, we classify any non-streaming accesses as being caused by an irregular access pattern.

4) Finally, it may happen that we do not find any non-zero stride for a particular instruction, either because the memory instruction accesses a fixed location, or because the instruction is not enclosed in any loop. Our data flow analysis is performed only intra-procedurally at this time, and we found many instances, especially in `C++` code, where frequently executed routines did not contain any loops, but they accessed memory locations determined by the function input parameters. Currently, we cannot classify the non-streaming accesses produced by these instructions using static analysis. This is a limitation of our current implementation, not a limitation of the approach. Understanding if these accesses are strided or irregular requires at least partial inter-procedural analysis. For completeness, in Section 6, we classify such accesses as having no stride.

[2]*constant* in this context means that the stride can be determined statically, it is not depended on run-time values.

We use the above analysis to classify non-streaming memory accesses into four main categories: 1) accesses with irregular strides; 2) accesses with large strides, with the added sub-category of symbolic strides; 3) streaming accesses with high concurrency levels produced by an innermost loop; 4) streaming accesses with high concurrency levels produced by an outer loop. In addition, we also classify streaming accesses with a concurrency level higher than 16 into either category (3) or category (4), depending on which loop level produces the smallest stride.

6. EXPERIMENTAL RESULTS

We applied our post-mortem analysis to the SPEC CPU 2006 benchmark suite simulation results. Figure 4 presents the distribution of prefetch unfriendly accesses with respect to the DC prefetcher, for each of the benchmarks. As before, the labels on the `x` axis show the benchmark index and the name of the input file. Benchmark names are listed in Table 2. Most of the bin labels should be self explanatory. The bins labeled as `17+ Inner` and `17+ Outer` correspond to streaming accesses with high concurrency levels produced by the innermost loop or by an outer loop, respectively.

Figure 4 shows the fraction of non-prefetchable L1 cache misses for each of the categories described in Section 5. The stacked histograms do not sum up to 100% this time. The difference to 100% is represented by cache misses that were determined to be prefetcher friendly. Many benchmarks include a significant fraction of accesses for which our intra-procedural data flow analysis could not find an access stride.

If we focus on the accesses for which strides could be determined, we notice that many benchmarks have a large fraction of L1 cache misses that are produced by irregular or indirect accesses. This is not such a surprise when we look at the list of integer benchmarks included in the suite: `400.perlbench` is a perl interpreter, `401.bzip2` is a compression program, `403.gcc` is a C compiler, `429.mcf` implements a vehicle-depot scheduling algorithm, and `473.astar` implements a path finding algorithm for a computer game. Even among the floating point benchmarks we find a good fraction of irregular memory access patterns. Benchmark `435.gromacs` is a molecular dynamic code that uses a neighbor list method to compute interactions between nearby particles. While accesses to the neighbor list itself are strided, accesses to the actual particle data are indirect and irregular.

Somewhat surprisingly, only three benchmarks have any significant number of streaming accesses with high concurrency levels produces by an outer loop: `434.zeusmp` - a magnetohydrodynamics code, `447.dealII` - a partial differential equations solver, and `459.GemsFDTD` - a computational electromagnetic code. We think that such cases are not particularly interesting as targets for prefetching optimizations, because the access patterns that generate them are also characterized by long spatial reuse, and optimizing for data reuse should improve streaming behavior as well.

On the other hand, accesses with high concurrency levels produced by an inner loop are interesting for further analysis and optimization, because they capture situations

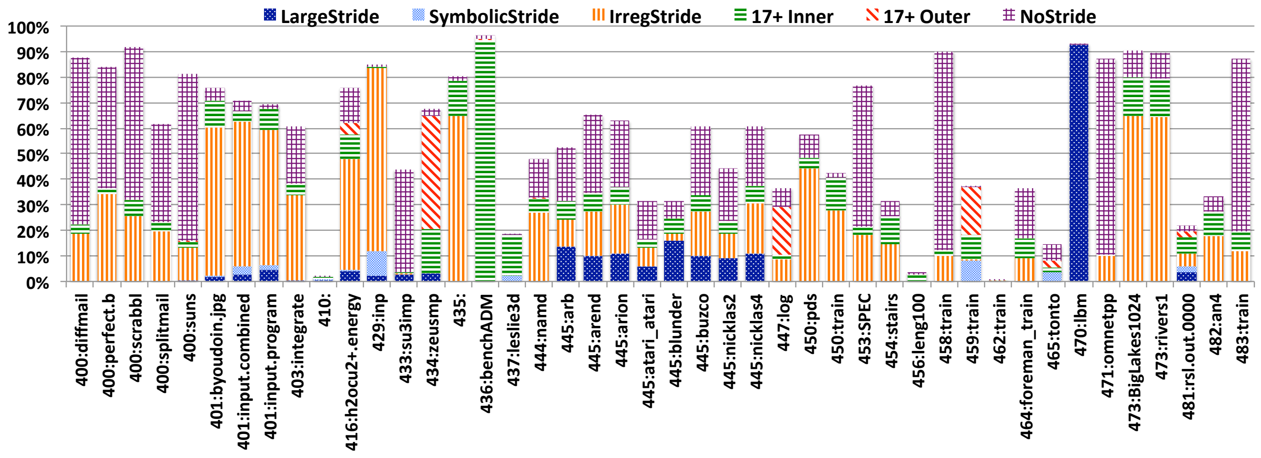


Figure 4: Distribution of prefetch unfriendly accesses with respect to the DC prefetcher.

where applications stream over many data arrays in an inner loop. All accesses are performed with a small stride and they should exhibit good spatial locality. However, streaming over too many arrays at the same time makes hardware prefetchers ineffective, which may significantly impact performance. Such cases can be improved either by data layout transformations such as fusing several arrays into one array, interleaving their elements, or by code transformations such as loop splitting, with each loop streaming over only some of the arrays. Each of these approaches can have negative side effects, so they should be applied on a case by case basis. Array fusion works best when the arrays that are fused are always accessed together. If we fuse arrays that are not always accessed by the same loops, we negatively impact spatial reuse and increase memory bandwidth use. Loop splitting should be applied when we do not create data reuse between the resulting loops, in other words, when the resulting loops access disjoint sets of arrays. We notice that benchmark `436.cactusADM` has a very large fraction of accesses with high inner concurrency levels. In Section 6.1 we attempt to improve its streaming behavior.

We also find several benchmarks that have a meaningful fraction of strided accesses where the stride is large. Benchmark `470.lbm` stands out, with a very high fraction of accesses characterized by a large stride. We want to take a closer look at this benchmark as well, because we do not know exactly what to expect. We observe that, in the case of the SPEC CPU benchmarks, only a small fraction of non-streamable accesses are associated with symbolic strides.

6.1 Tuning of benchmark 436.cactusADM

CactusADM is based on the open source Cactus problem solving environment, and on the computational kernel BenchADM. CactusADM solves the Einstein evolution equations, which describe how space-time curves in response to its matter content, using a set of ten coupled nonlinear partial differential equations [7]. Our analysis of the CactusADM results revealed that almost all of its `17+ Inner` accesses are produced by two loops in routine `Bench_StaggeredLeapfrog2`. This Fortran routine has 135 input arguments, and is called from within the Cactus framework. Around half of its input parameters are arrays, many of them three-dimensional arrays.

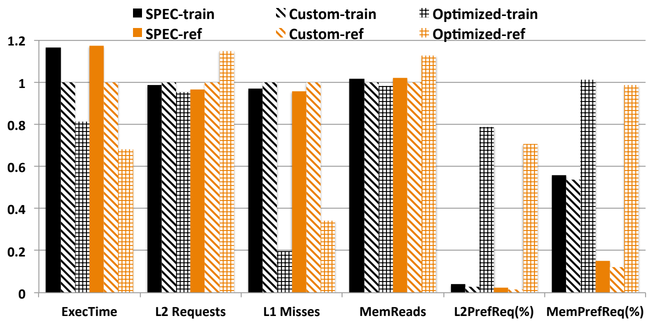
The Cactus framework auto generates a lot of the glue code that allocates and initializes data arrays, based on a custom input language. We could not determine how to modify the Cactus framework to do the layout transformations that we wanted. Instead, we wrote a custom wrapper code in C that allocates and initializes all the arrays expected by the Fortran routine, and then invokes the unmodified Fortran code, just as the Cactus code does. We call this version the CustomADM code.

Next, we created an optimized version of the code based on our analysis results. We fused 69 of the input arrays into 13 larger arrays, using their names[3] and their dimension sizes to guide our transformations. An additional six arrays locally declared in the Fortran routine were fused into a single array. We call this code the OptimizedADM version. We measured the performance of the three code versions, the original SPEC ADM code, the custom ADM code, and the optimized ADM code, using hardware performance counters.

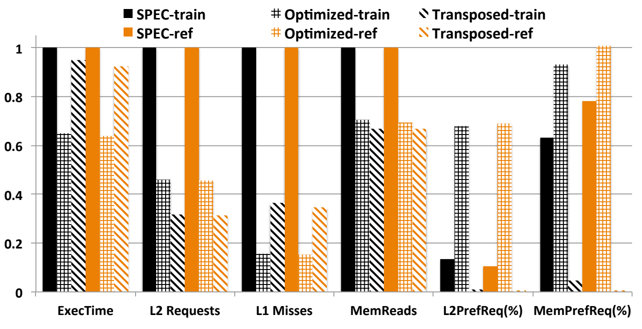
Figure 5a shows the results for both the `train` and the `ref` inputs. The first four metrics shown in the figure are normalized to the performance of the Custom version. The number of L2 prefetch requests is shown as a percentage of the total number of L2 requests for each version of the code. Similarly, the number of memory prefetch requests is shown as a percentage of the total number of memory read requests for each code version. We first compare the SPEC and the custom versions of the code, to show that the custom version is a good substitute for the SPEC Cactus code. While the execution time of the custom version is 14% shorter than the original execution time because some of the Cactus specific code is not executed anymore, the measured metrics show that the memory behaviors of the two versions are similar.

We then compare the optimized and the custom versions. The fraction of L2 requests initiated by the DC hardware prefetcher increases dramatically, from about 2% to about 79% for the `train` input, and to 71% for the `ref` input. As a result, we see a similarly significant drop in the number of L1 misses for the optimized version. The fraction of memory requests initiated by the MC prefetcher also increases from 54% with the `train` input and 12% with the `ref` input, to about 100% for the optimized version. This increase in pre-

[3]Related arrays were named with a common prefix.



(a) 436.CactusADM results



(b) 470.LBM results

Figure 5: Tuning results for benchmarks CactusADM and LBM with the `train` and `ref` inputs.

fetching coverage yields a close to 19% and 32% reduction in execution time for the two inputs, or a 23% and 47% speed-up, respectively. The prefetch optimizations yield a larger speed-up for the `ref` input because the original code's MC prefetching performance was lower with the larger input. The `ref` input results show also an increase of 12-15% in memory and L2 traffic for the optimized version due to superfluous prefetch requests, mentioned in §4.3.

6.2 Tuning of benchmark 470.lbm

The SPEC LBM benchmark implements the Lattice Boltzmann Method to simulate incompressible fluids. We analyzed the results obtained for the LBM benchmark and we found that all the prefetch unfriendly accesses with a large stride were produced by routine `LBM_performStreamCollide`. The benchmark performs sweeps over a three-dimensional lattice, implemented as a one-dimensional array in the SPEC version of the code. The state of each lattice site is defined by a set of double precision floating-point numbers that represent the distribution functions with respect to the discrete directions of velocities [18]. Thus, the lattice data structure in the `470.lbm` benchmark is implemented as an array of records with 20 double precision floating point fields.

Routine `LBM_performStreamCollide` performs a 19-point Stencil 3D calculation using separate source and destination lattices. Only some of the fields of the destination lattice grid are updated for each of the sites. Each lattice element is a record with 20 double precision floating point values, thus, 160 bytes in size. Although all the lattice sites are traversed in sequential order, the type of sparse updating performed by the LBM code, results in accesses with strides larger than a cache line (64 bytes), and therefore, are not recognized as streaming accesses by the AMD 10H DC prefetcher.

To improve streaming behavior, we created an optimized version by splitting the array of records into three separate arrays of smaller sized records, each record's size being less than or equal to one cache line. Performance results for the `train` and `ref` inputs are summarized in Figure 5b. The first four metrics are normalized to the performance of the SPEC version. The number of L2 prefetch requests is shown as a percentage of the total number of L2 requests for each version of the code. Similarly, the number of memory prefetch requests is shown as a percentage of the total number of memory read requests for each code version. LBM's `ref` and `train` inputs differ mainly in the number of executed time steps. Thus, the performance profiles of the two inputs are very similar. The comments below apply to both.

We compare the SPEC and the Optimized versions of the code. The optimized version's execution time is 35% smaller than the SPEC LBM version, or in other words, it achieves a 54% speed-up. However, we immediately notice that the optimized version has much better data locality. The number of L2 requests and the number of memory reads dropped by 54% and 30%, respectively. We also observe a significant increase in the fraction of L2 requests that are initiated by the DC hardware prefetcher, on top of the overall reduction in L2 requests. Unfortunately, the transformation that improved prefetching effectiveness also improved data locality, and we cannot separate or evaluate their effects individually.

Instead, we created a new version of the code, where we split each state variable into its own array. The transformation consists of transposing the array of records into a record of arrays. Hence, we named this code the Transposed version. Naturally, this transformation produces the best data locality among all versions, as can be seen by looking at the counts of L2 requests and memory reads in Figure 5b. However, creating a separate array for each state variable increases the code's streaming concurrency beyond what the two AMD 10H hardware prefetchers can handle. As a result, this code version exhibits almost no DC or MC prefetches. We also notice that despite the incrementally improved data locality, the transposed version runs much slower, 46.5% slower than the optimized version.

7. CONCLUSIONS

Streaming hardware prefetchers can have an important beneficial effect on application performance, if applications traverse memory with the right access patterns. However, these hardware structures are largely hidden from users. Unlike cache memories for which a large number of analysis and tuning techniques have been proposed and studied, there has been little previous work that focused on application analysis techniques specifically targeted at identifying tuning opportunities for the hardware prefetchers.

In this paper, we introduced the metric *streaming concurrency* to characterize the number of parallel, logical data streams in applications. Our experiments show that the *streaming concurrency* metric explains well the number of prefetch requests initiated by the two AMD 10H hardware prefetchers for the SPEC CPU2006 benchmark suite. While not the focus of this paper, the *streaming concurrency* metric can also be used to guide the design of the hardware structures so that they handle well a particular workload.

We described an approach based on static analysis and simulation to understand if the memory access patterns of applications are amenable to hardware prefetching, and to identify opportunities for improving their prefetch friendliness. We identified four situations that make the streaming hardware prefetchers ineffective and we provided insight into the code transformations that are needed to increase the prefetching effectiveness in each of the four situations. We used this insight to tune the prefetching performance of two SPEC CPU2006 benchmarks. While some of the performance gains observed in Section 6 can be attributed to data locality improvements, our tuning results show that it is important to write prefetch friendly code whenever possible, to minimize unfulfilled performance.

We do not advocate prefetching optimizations at any cost, for example at the expense of data locality. A balanced approach is needed in most cases. Moreover, increasing data reuse and eliminating data movement between the different levels of the memory hierarchy altogether, can also improve the energy efficiency of an application. Our tuning results with the LBM benchmark show, however, that we should not pursue data reuse improvements exclusively, at the cost of prefetching effectiveness, because such an approach can negatively impact performance.

8. ACKNOWLEDGMENTS

This research is sponsored, in part, by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy. The paper has been coauthored by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

9. REFERENCES

[1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 176–186, New York, NY, USA, 1991. ACM.

[2] B. Bennett and V. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[3] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing And Systems*, pages 617–662, 2001.

[4] 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.

[5] 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.

[6] 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.

[7] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comp. Arch. News*, 34(4):1–17, Sept. 2006.

[8] 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.

[9] 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.

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

[11] A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 66 –75, march 2010.

[12] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13. ACM Press, 2004.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

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

[15] 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.

[16] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[17] 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.

[18] J. Wilke, T. Pohl, M. Kowarschik, and U. Rüde. Cache performance optimizations for parallel lattice boltzmann codes. In *Euro-Par*, pages 441–450, 2003.

[19] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.