An Adaptive Data Prefetcher for High-Performance Processors

Yong Chen
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
yong.chen@iit.edu

Huaiyu Zhu
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
hzhu12@iit.edu

Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
sun@iit.edu

Abstract—**While computing speed continues increasing rapidly, data-access technology is lagging behind. Data-access delay, not the processor speed, becomes the leading performance bottleneck of high-end/high-performance computing. Prefetching is an effective solution to masking the gap between computing speed and data-access speed. Existing works of prefetching, however, are very conservative in general, due to the computing power consumption concern of the past. They suffer in effectiveness especially when applications' access pattern changes. In this study, we propose an Algorithm-level Feedback-controlled Adaptive (AFA) data prefetcher to address these issues. The AFA prefetcher is based on the Data-Access History Cache, a hardware structure that is specifically designed for data prefetching. It provides an algorithm-level adaptation and is capable of dynamically adapting to appropriate prefetching algorithms at runtime. We have conducted extensive simulation testing with SimpleScalar simulator to validate the design and to illustrate the performance gain. The simulation results show that AFA prefetcher is effective and achieves considerable IPC (Instructions Per Cycle) improvement in average.**

Keywords-data prefetching; high-performance processors; computer architecture; memory wall

I. INTRODUCTION

As semiconductor process technology and micro-architecture evolve, the processor cycle times have been significantly reduced in the past decades. However, compared to processor performance improvement, data-access performance (latency and bandwidth) improvement has been at snail's pace. The memory speed has only increased by roughly 9% each year over the past two decades, which is significantly lower than the improvement speed of nearly 50% per year for processor performance [12]. This performance disparity between processor and memory is predicted to continually expand in next decades [12]. Multi-level cache hierarchy architectures have been the primary solution to avoiding large performance loss due to long memory-access delays. However, cache memories are designed based on data access locality. When applications lack data access locality due to non-contiguous data accesses, multi-level cache memory hierarchy does not work well.

Data prefetching is one of the basic techniques associated with multi-level cache hierarchy to bridge the performance gap between processor and memory. The basic idea of data prefetching is observing data access patterns, speculating future access addresses and fetching the data from predicted addresses in advance. Data prefetching can at least partially overcome the limitations of cache memories, and reduce long memory access latency by overlapping computation and data accesses. Intensive studies have been conducted and many strategies have been proposed for data prefetching in the past [4][5][6][7][8][14][15][17][23][26][28]. Prefetching has been adopted in production and found to be very effective for special applications with simple data access patterns, such as data streaming [15][8]. But in general, current prefetching techniques are very limited. Software solutions are too slow for cache level prefetching; whereas hardware prefetchers are static in nature and cannot change with the data access patterns of the applications. Previous studies show that there is no enough effort to support hardware dynamic adaptation among different strategies depending on the runtime application characteristics [3]. The fact is that data prefetching is application and data access pattern dependent. There is no single universal prefetching algorithm suitable for all applications at runtime. A general and effective data prefetcher must be dynamic in nature.

In this study, we propose an Algorithm-level Feedback-controlled Adaptive (or AFA in short) data prefetcher to exploit a dynamic and adaptive prefetching methodology based on the recently proposed generic prefetching structure, Data-Access History Cache (DAHC) [5], and runtime feedback collected by hardware counters. DAHC is a new cache structure designed for data prefetching. It is capable of effectively tracking data-access history and maintaining correlations of both data access address stream and instruction address stream. It can be used for efficient implementation of many history-based data prefetching algorithms [5]. Hardware counters gain considerable attention in recent years and are becoming more and more important for improving performance of contemporary architecture, operating system and applications [22][27][30]. This study exploits this trend and assumes hardware counters are available within a data prefetcher that generally resides on the lowest cache level. Based on DAHC and available hardware counters, the AFA data prefetcher is able to recognize distinct data-access patterns and adapts to the corresponding appropriate prefetching algorithms at runtime. This adaptation methodology is significantly better than

conventional static prefetching strategies. It improves the prefetching effectiveness which in turn improves the overall performance. AFA prefetcher does consume some transistors. However, the hardware chip space and the number of transistors integrated on chip are not limitations for current processor architectures. Trading chip space for lower access latency is a current trend [16][29]. AFA prefetcher and DAHC follow the trend.

The rest of this paper is organized as follows. Section II briefly reviews the DAHC structure. Section III presents the proposed AFA prefetcher design and discusses implementation related issues. Section IV presents the simulation environment and simulation results. Section V discusses related work, and finally Section VI concludes this study.

II. Data-Access History Cache

To fully exploit the benefits of data prefetching and to focus on reducing data-access latency to achieve a high sustained performance instead of building extensive compute units to achieve a high peak performance, we proposed a generic prefetching-dedicated cache structure, named *Data-Access History Cache (DAHC)* [5]. The DAHC serves as a fundamental structure dedicated to data prefetching. The DAHC behaves as a cache for recent reference information instead of as a traditional cache for either instructions or data. Theoretically, it is capable of supporting any history-based prefetching algorithms.

The design rationale of DAHC is that history-based prefetching algorithms must rely on correlations among either instruction address stream or data address stream, or both. Thus, DAHC is designed to have three hardware tables: one data-access history table (DAH table) and two index tables, program counter index table (PIT) and address index table (AIT). The DAH table accommodates history details, while PIT and AIT maintain correlations from instruction and data address stream viewpoints respectively. Various prefetching algorithms [5] thus can access these two tables to obtain the required correlation as necessary. Fig. 1 illustrates the general design of DAHC and a high-level view of how it can be applied to support various prefetching algorithms.

Fig. 2 illustrates the detailed structure of DAHC through an example. The DAH table consists of PC (program counter), chain_PC, Addr, chain_Addr and State fields. PC and Addr fields store the instruction address and data address separately. The chain_PC and chain_Addr point to an entry where the last access from the same instruction or the last access of the same address is located. Therefore, chain_PC and chain_Addr connect all accesses from the instruction stream and data stream perspectives. This design offers the fundamental mechanism to detect potential correlations and access patterns. The State field maintains state machine status used in prefetching algorithms. The PIT table has two fields, PC and Index. The PC field represents the instruction address, which is a unique index in this table. The Index field records the entry of the latest data access in the DAH table from the instruction stored in the correspondent PC field. It is the connection between the PIT and the DAH tables. The address index table is similarly defined. For instance, in Fig. 2, the DAH table captured four data accesses, three of them issued by instruction 403C20 (stored in the PC field) and one by instruction 4010D8. The instruction 403C20 accessed data at address 7FFF8000, 7FFF8004 and 7FFF800C in sequence, which is shown through the Addr and chain_PC fields. The instruction 403C20 and 4010D8 are also stored in the PIT table, and the corresponding index field tracks the latest access from the DAH table, which are entry 3 and 1 respectively. The AIT table keeps each accessed address and the latest entry, as shown in the bottom left of the figure, thus connecting all the data accesses on the basis of the address stream.

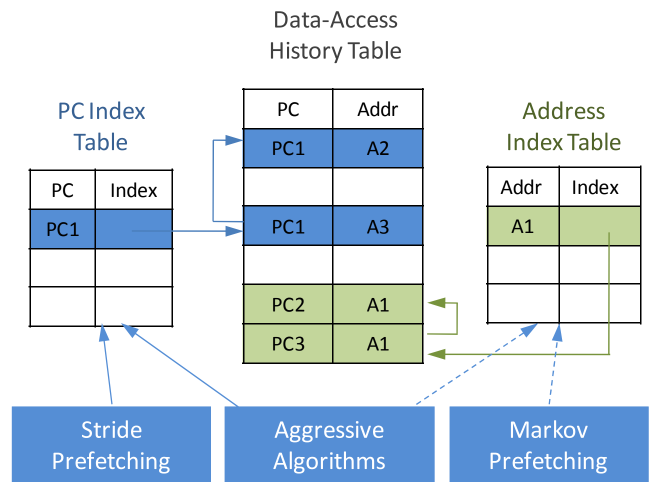
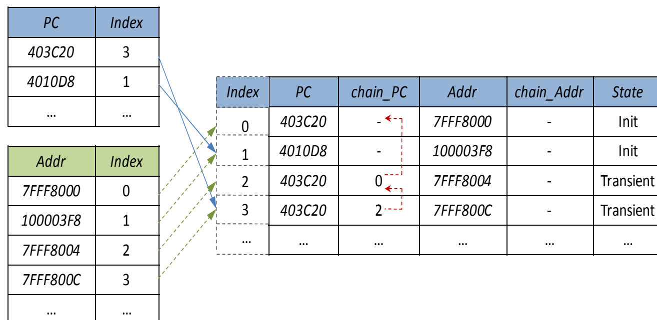


Figure 1. DAHC general design and a high-level view.

The DAHC provides a prototype design of a prefetching-dedicated structure. It works as a cache for data access information compared with conventional cache for either instructions or data. The DAHC can be placed at different memory hierarchy levels for various desired data prefetching. For instance, it can be used to track all accesses to first level cache and to serve as an L1 cache prefetcher. It can also be placed at the second level cache and to serves as an L2 cache prefetcher only. The straightforward design makes the implementation uncomplicated. The implementation of the DAHC is a specialized physical cache, like victim cache [13] or trace cache [20]. The PIT and AIT tables can be implemented with any associativity such as 2-way or 4-way. Since the index tables usually have less valid entries than the DAH table, it is unlikely that some entry is replaced due to a conflict miss [5]. Even if a conflict miss occurs, it does not affect the correctness except discarding certain access history. The DAH table can be implemented with a special structure where each entry can be located by using its index. The logic to fill/update the DAHC comes from the cache controller. The cache controller traps data accesses at the monitored level and

keeps a copy of the access information in the DAHC. Note that DAHC design is general and it does not imply any restriction to the system environment. It works in CMP or SMT environment, as well as in an environment where multiple applications are running concurrently.



PC	Index
403C20	3
4010D8	1
...	...

Addr	Index
7FFF8000	0
100003F8	1
7FFF8004	2
7FFF800C	3
...	...

Index	PC	chain_PC	Addr	chain_Addr	State
0	403C20	-	7FFF8000	-	Init
1	4010D8	-	100003F8	-	Init
2	403C20	0	7FFF8004	-	Transient
3	403C20	2	7FFF800C	-	Transient
...	...	...	...	...	...

Figure 2. DAHC design: PIT, AIT and DAH tables.

III. ALGORITHM-LEVEL FEEDBACK-CONTROLLED ADAPTIVE DATA PREFETCHER

In this section, we present the design of an Algorithm-level Feedback-controlled Adaptive data prefetcher. This proposed AFA prefetcher leverages the powerful functionality provided by DAHC, supports multiple prefetching algorithms and dynamically adapts to those algorithms that perform well at runtime. The essential idea is using runtime feedback and evaluation to direct the dynamic adaptation. We first introduce the evaluation metrics, and then discuss the implementation and the methodology of directing adaptation.

A. Evaluation Metrics

While extensive studies exist in prefetching, few studies present a formalized metric to evaluate the effectiveness of prefetching algorithms. We analyze and sort out the essential and most critical criteria to model prefetching evaluation and present a formal definition in this study. These metrics provide a comprehensive evaluation of hardware prefetching methodologies. These metrics can be independently used to evaluate a prefetcher in addition to commonly used metrics, such as an IPC speedup metric.

1) Prefetch Precision: The first and widely-adopted metric is termed *prefetching precision* or *prefetching accuracy*. This metric characterizes the percentage of prefetches that are actually accessed by demand requests, thus reflecting how accurate the prefetch requests and the prefetching algorithms are. We present a formal definition of prefetching precision as follows.

Definition 1. *Prefetching precision is defined as the ratio between the number of distinct prefetched cache lines that are accessed by at least one demand request after being prefetched in and before being replaced out over the number of total prefetched cache lines.*

By this definition, the prefetching precision models the *accuracy* of the prefetcher, i.e. the percent of *useful* cache lines in the *overall* cache lines prefetched. Note that we define a useful prefetch as being accessed at least once when the prefetched cache line resides in the prefetch destination. Therefore, a repeated access to that cache line in the current lifecycle does not account into the total number of useful prefetches. However, if a cache line is prefetched again into the destination after being displaced, and gets hit, this scenario will contribute to one useful prefetch. We refer these cache lines brought in by prefetch and accessed by demand requests as *prefetch hits*, in contrast with *demand hits*, those lines fetched by demand and hit again by other requests.

The prefetch precision should be considered as the most critical metric to evaluate or direct prefetch adaptation, as it describes the *cost-efficiency* of a prefetcher well. Taking prefetch precision into consideration, an aggressive but not accurate enough prefetcher should be largely avoided because it might produce a large number of useless prefetches, which significantly wastes resources, such as power and cache line slots. Instead, this metric favors a prefetcher with high-confidence. The prefetch precision metric suggests that the prefetcher should focus on identifying the correct access pattern and make a highly-accurate prediction. Such an approach maximizes the hardware investment on the prefetcher and achieves a high cost-efficiency. An ideal prefetcher will produce a prefetching precision with value 1. In practice, the prefetching precision has a range from 0 to 1.

Though the prefetch precision is critical and straightforward, it merely describes one aspect of the problem under study – the prefetch precision does not quantify how effective the prefetcher is, i.e. how many misses among the overall misses are hidden. The next metric we formalize addresses this limitation.

2) Prefetch Coverage: The prefetch coverage metric is introduced to complement prefetch precision and quantify the other aspect of how well a prefetcher works. We formalize the prefetch coverage definition as follows.

Definition 2. *Prefetch coverage is defined as the ratio of the number of misses reduced due to prefetches over the total number of misses that will occur without prefetching.*

As the definition states, the prefetch coverage focuses on quantifying the ratio of the misses reduced, i.e. how wide a prefetcher covers the demand misses that are supposed to occur without the assistance of prefetching. A highly-accurate prefetcher does not necessarily provide a wide coverage. This is because such a prefetcher could be very conservative and takes action only when the prefetcher has a high-confidence prediction. This conservativeness results in a high precision but the effectiveness in terms of miss reduction ratio is low. The vice versa holds as well, i.e. a prefetcher with wide coverage is not necessarily highly-accurate since the prefetcher could be very aggressive (such as with a large prefetch degree) to improve the coverage while sacrificing the precision. In essence, the prefetch coverage and prefetch precision are complementary to each

other, and together they quantify the effectiveness of a prefetcher from two aspects.

3) Prefetch Pollution: While prefetch precision and prefetch coverage can reflect the prefetch effectiveness, or the positive side, of a prefetching algorithm well, they do not characterize the negative side of an algorithm. Cache pollution [28] is considered a critical down side of prefetching. When a cache line that is replaced by a prefetched line is later accessed by a demand request, cache pollution occurs. Such a cache miss will not happen without the interference of prefetching. This scenario, cache pollution, is referred to as a negative side-effect of prefetching. We present a formal definition to describe prefetch pollution.

Definition 3. *Prefetch pollution is defined as a ratio of the number of additional demand misses caused by prefetching that will not occur without prefetch interference over the number of misses that will occur without prefetching.*

According to this definition, prefetch pollution quantifies the percent of extra demand misses due to prefetches, which means that those demand misses will not occur if prefetching is not adopted. The occurrence of these misses is due to the limited cache size and the replacement of useful cache lines by prefetched cache lines. Together with prefetch precision and prefetch coverage, the prefetch pollution completes a three-tuple, *(precision, coverage, pollution)* or *PCP* in short, to evaluate a prefetching algorithm. These three metrics are complementary to each other, and assess an algorithm from both positive and negative aspects. Some literatures separate other metrics, such as lateness [25]. We observed that these metrics are well covered in the 3-tuple PCP metric. A separation of these additional metrics might be helpful, but might also cause confusion. In this study, the proposed dynamic adaptation is based on the 3-tuple PCP metric.

B. Evaluation Metrics: On-the-Road

We have presented the formal definitions of a PCP metric to evaluate and direct a prefetcher in the previous section. We discuss the hardware design and realization of these metrics in the proposed AFA prefetcher in this section.

1) Realizing Prefetch Precision Metric: To realize the prefetch precision metric, the AFA prefetcher utilizes two statistics counters for each evaluated algorithm, one counter for the prefetch hits, and one counter for overall prefetches. We refer these two counters as prefetch_hits and prefetch_total. In addition, to collect the statistic of prefetch hits, we need to distinguish the cache lines prefetched from demanded. This requirement results in a major hardware storage budget. We assume each cache line in the prefetch destination (L2 cache in this study) has one extra prefetch bit to represent whether this line is prefetched or fetched for each evaluated algorithm. When a cache line is prefetched into destination, this prefetch bit is set. If this cache line is ever accessed during its lifetime in the cache (after being prefetched and before being displaced), the prefetch_hits counter is increased and the prefetch bit is reset. By this way, the prefetched cache line is not counted as multiple hits even when it is accessed multiple times, which is consistent with the definition. A simple reason behind this decision is that the first hit acts like a regular demand request and fetches in data, and the future accesses will hit in cache. The actual saving of the prefetching is the first access. If a cache line is brought in by a normal demand request, the prefetch bit is not set. The combinatorial logic to maintain this prefetch bit, set and reset, is trivival, and the hardware implementation of this logic is not complicated. Note that if a cache line is prefetched by multiple algorithms, the corresponding prefetch bits will be set and a hit of this cache line will attribute to the statistics of each corresponding prefetcher.

2) Realizing Prefetch Coverage Metric: The prefetch_hits counter discussed above can also be used in calculating the prefetch coverage. This is because that the statistics the prefetch_hits counter collects is the number of misses reduced due to prefetches. To compute the prefetch coverage, the AFA prefetcher needs another counter − the number of overall misses that will occur without prefetching. The AFA prefetcher utilizes another acounter, demand_misses counter, to collect the number of misses that occurs even with prefetching. The prefetch_hits counter represents the number of misses saved by prefetch, and the demand_misses counter represents the number of misses that still occur. The prefetch coverage is computed as:

$$prefetch_coverage = \frac{prefetch_hits}{prefetch_hits + demand_misses}$$

3) Realizing Prefetch Pollution Metric: It is more challenging to collect the prefetch pollution statistics than to collect prefetch precision and coverage. The reason is that we never know whether the replaced cache line due to a prefetch will be used in future or not. An optimal solution to collecting the prefetch pollution metric is tracing down all these cache lines and detecting whether any future requests will access these lines. If such a cache line is detected, which means that this cache line is replaced out due to a prefetch but is needed by a demand request, a case of cache pollution is detected. However, such an optimal solution will require infinite-size storage to keep all past cache lines replaced out due to prefetches. This approach is not feasible in practice. Motivated from existing studies [1][18][25], the AFA prefetcher utilizes a Bloom filter [1] to estimate the percentage of cache pollution.

Suppose the cache line size is 64B, and a cache block address is 26 bits. The pollution filter splits 26 bits into two parts, high-order 13 bits and low-order 13 bits. These two parts are fed into an XOR logical unit, and a filtered address, with 13 bits, is the output. This filtered address is used to index a bit vector and set the corresponding bit in

the vector. The AFA prefetcher uses this filter to estimate the pollution. It tracks each cache line that is replaced out due to a prefetch and feeds this cache line address into the filter. A corresponding bit of the bit vector is set. It also feeds cache miss addresses into the filter, and if the corresponding bit is set, the AFA prefetcher estimates this cache line was in the cache but was replaced out due to a prefetch. After a cache pollution is detected, the corresponding bit is reset, as the cache line is fetched back into cache. The AFA prefetcher uses a pollution counter for each evaluated algorithm to accumulate the prefetch pollution statistics. The hardware cost of Bloom filters is discussed in Section III-E.

C. Metrics Collection

The AFA prefetcher periodically collects the PCP metric discussed above in order to make adaptation decision. The adaptive prefetching mechanism is designed to have two phases, *metrics collecting phase* and *stably prefetching phase*. In the collecting phase, all supported prefetching algorithms are enabled, and the statistics of each prefetching algorithm are tracked and collected. In the end of collecting phase, the PCP metric is computed for prefetching algorithm evaluation. In the stably prefetching phase, only the adaptively selected algorithm will be running, and all counters and pollution estimator are cleared and turned off. The decision to choose the working algorithm is discussed in the following subsection.

The switch between these two phases is controlled by a *phase timer*. There are many potential ways for measuring the time and providing the phase timer, such as utilizing CPU cycles, issued instructions, issued load/store instructions or load/store misses. We choose the approach that views each cache miss as one time tick and accumulates to measure the time and provides the phase control. This is a feasible design choice to control the adaptive prefetcher behavior because the number of cache misses can fairly represent how the prefetcher should react. In addition, this design is considered better than utilizing cycles or issued instructions, as those numbers could be huge and increase rapidly. Utilizing cache misses as time ticks is a much simpler way for the AFA prefetcher. The AFA prefetcher is empirically configured with a collecting phase as $1/8^{th}$ of the stably prefetching phase, which means it collects statistics and makes adaptive selection decision in one unit of time, and prefetch with selected algorithms for eight units of time.

D. Adaptive Selection

After collecting statistics and computing the metrics, the AFA prefetcher makes the decision to adaptively select the suitable prefetching algorithms. The decision is based on the evaluation of the performance of each prefetching algorithm, indicated by the 3-tuple metric, the precision, coverage and pollution. This evaluation is not complicated – simply done by comparing the runtime statistics against a preset threshold and classifying them as either high or low. If it is above the threshold, the prefetcher classifies the statistics as high. In contrast, if it is below the threshold, the prefetcher classifies it as low. In our simulation experiments, the threshold to distinguish a high/low prefetch precision, coverage and pollution are preset as 0.70, 0.3, and 0.2 respectively based on empirical experience. In practice, these thresholds can be measured and determined in advance for any specific architecture. It can also be tuned dynamically at runtime. Fig. 3 illustrates a table of eight levels of prefetching algorithm performance that can be recognized by the prefetcher.

	Precision	Coverage	Pollution
Level 0	H	H	L
Level 1	H	L	L
Level 2	L	H	L
Level 3	L	L	L
Level 4	H	H	H
Level 5	L	H	H
Level 6	H	L	H
Level 7	L	L	H

Figure 3. Prefetching algorithm performance table.

Based on the prefetching algorithm evaluation performance table, the AFA prefetcher is able to identify and choose best prefetching algorithms dynamically. In our current study, we propose and analyze two different mechanisms, best-strategy adaptive selection and multi-strategy adaptive selection, to select the algorithms adaptively.

The best-strategy adaptive selection always outputs the one performing best in the statistics collecting phase. This decision is made based on the algorithm evaluation and the performance table – the lowest level is assigned to have the highest priority. Within the same level, the precision is assigned to have the highest preference, while the coverage has the medium and the pollution metric has the lowest preference. This means that if the AFA prefetcher sees multiple algorithms falling into multiple levels, the prefetcher chooses the lowest level algorithms as the candidate. If the prefetcher detects multiple candidates sharing the same lowest level, it favors the one with the highest precision.

The best-strategy adaptation works by choosing the best strategy according to the defined policy (performance table and preference assignment) out of all supported algorithms. However, a potential limitation is that it only chooses one algorithm even if multiple strategies are performing well and can sometimes complement each other. In addition, this strategy always outputs one "relatively best" strategy, even though the best strategy might not work well enough at certain circumstances. Based on these observations, we introduce another adaptation strategy, multi-strategy adaptive selection, that chooses multiple optimal strategies

according to the evaluation and performance table. This adaptation strategy uses the level as the selection criteria. For instance, if level 0 and level 1 are configured as the adaptation criteria, then the AFA prefetcher dynamically chooses all of these algorithms that fall into these levels, and use them in the stably prefetching phase. This adaptive strategy can also control the quality of the selection. If none of the algorithms satisfies the specified criteria, the prefetcher does not have any algorithm performing in the prefetching phase, until the algorithms are evaluated again in the next collecting phase. The selection criteria are preset, for instance, as level 0, 1 and 2 in our current simulation experiments.

E. Hardware Cost

As discussed in previous subsections, the AFA prefetcher needs three counters for each evaluated algorithm and two counters for all algorithms to collect the required statistics in order to direct the adaptation. Each counter can be implemented with a 32-bit register. The overall required storage for counters will be 56 bytes, if assuming to support four distinct algorithms simultaneously. In addition to the counters, the AFA prefetcher needs one cache pollution estimator for each supported algorithm. The pollution estimator requires 2^{13} bits of storage for the bit vector. Therefore, the estimator consumes 1KB for each supported algorithm. The cache structure needs a slight modification to support adaptive selection of prefetching algorithm as discussed previously. The modification is one bit for each supported algorithm. For a typical 1MB L2 cache with 64 bytes cache line, it has 16,384 cache lines. To support one prefetching algorithm, the additional hardware cost will be 16,384 bits or 2KB. Therefore, as a normal case, to support adaptation among four algorithms, the overall hardware cost will be around 12KB. This hardware budget is trivial – as only around 1% compared to a regular 1MB L2 cache. However, as the simulation verifies, the adaptive prefetching can considerably reduce cache misses and improve the overall system performance.

As discussed partially in the previous section, the combinatorial logic to realize the proposed adaptive prefetching is not complicated as well. The major required combinatorial logic resides in maintaining the prefetch bits within the cache line, maintaining statistics counters, filtering through prefetch pollution estimator, and adapting prefetching algorithms via the performance table. Maintaining prefetch bits is straightforward because it merely requires set/unset the corresponding bit according to whether a cache line is brought in due to a specific algorithm. Maintaining counters is a straightforward logic too. Filtering is slightly complicated, but as we show with the estimator, the hardware state machine can be described effortlessly. Adapting the algorithm mainly needs comparison logic, which can be implemented easily as well.

IV. SIMULATION AND PERFORMANCE ANALYSIS

We have carried out simulation experiments to study the feasibility of the proposed AFA prefetcher and analyze the potential performance impact. Stream prefetching [7][15], stride prefetching [4][8], Markov prefetching [14] and MLDT prefetching [26] algorithms are selected for simulation. This section discusses simulation details and presents the analytical results.

A. Simulation Methodology

We have enhanced SimpleScalar simulator [2] with the DAHC [5] and the AFA prefetcher for the simulation verification and analysis. SimpleScalar tool set provides a detailed and high-performance simulation of modern processors. It takes binaries compiled for SimpleScalar architecture as input and simulates their execution on provided processor simulators. It has several different execution-driven processor simulators, ranging from extremely fast functional simulator to a detailed and out-of-order issue simulator [2].

We have chosen the most detailed, *sim-outorder simulator*, for our experiments. Fig. 4 shows the enhanced SimpleScalar simulator architecture. We have added two primary modules, DAHC module and AFA prefetcher module. The DAHC module simulates the functionality of the DAHC, as explained in [5]. The AFA prefetcher module implements the adaptive prefetching logic and four supported prefetching algorithms, stream, stride, Markov and MLDT. We have modified the cache line structure for identifying whether a cache line is brought in due to a prefetch, and which algorithm brings it in. We have created the required counters and the pollution estimator to simulate the evaluation as well. The combinatorial logic was simulated to compare the evaluation of each algorithm, and outputs the dynamically chosen suitable algorithms, with both best-strategy and multi-strategy selections. The multi-strategy selection was configured as level 0, level 1 and level 2 algorithms.

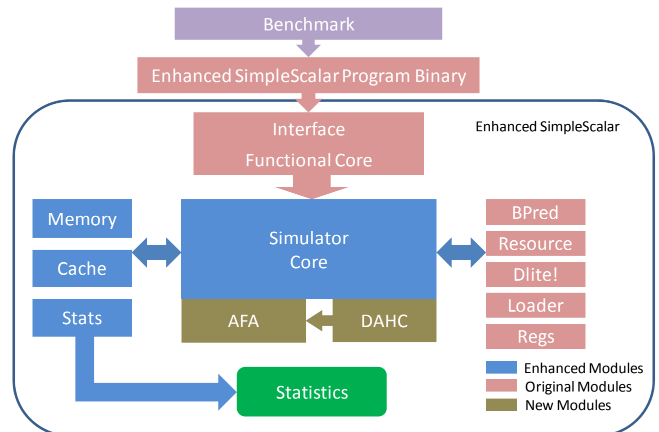


Figure 4. Simulation architecture of AFA prefetcher.

The AFA prefetcher is simulated to have an additional prefetch queue to store the prefetch requests. When the load/store issuing bandwidth is available after issuing demand requests, the prefetcher starts issuing the requests from the prefetch queue. If a newly generated prefetch request is already in the prefetch queue, the new request is simply dropped. If the prefetch queue is full, the new requests replace the old requests. The handling of prefetch requests is similar to the handling of regular load requests, with a slight difference that the effective address is computed based on prefetching algorithm, and any exceptions/faults generated by prefetches are discarded and the previous states are restored.

B. Simulation Setup

We use the Alpha-ISA and configure the simulator as a 4-way issue and 256-entry RUU processor. The instruction cache and data cache are split. L1 data cache is configured as 32KB 2-way with 64B cache line size. The latency is 2 cycles. L2 data cache is configured as 1MB 4-way with 64B cache line size. The latency of L2 cache is 12 CPU cycles. DAHC is configured with 1024 entries. We assume each DAHC access costs one CPU cycle. This should be a reasonable assumption for a fairly small cache. The AFA prefetch queue is configured with 512 entries. The stream prefetching algorithm is configured to support four streams. The prefetch degree for all prefetching algorithms is configured as eight. The prefetch distance for strided, Markov and MLDT is configured as four. Table I lists the configuration of the simulator in our simulation tests.

TABLE I. SIMULATOR CONFIGURATION

Issue width	4
Load store queue	64 entries
RUU size	256 entries
L1 D-cache	32KB, 2-way set associative, 64 byte line, 2 cycle hit time
L1 I-cache	32KB, 2-way set associative, 64 byte line, 1 cycle hit time
L2 Unified-cache	1MB, 4-way set associative, 64 byte line, 12 cycle hit time
Memory latency	120 cycles
DAHC	1024 entries
AFA queue	512 entries

C. Simulation Results

We have performed a series of SPEC-CPU2000 benchmarks [32] simulation for performance evaluation. We fast forwarded the first 100M instructions and simulated the following 200M instructions to analyze the result. Twenty-one out of total twenty-six benchmarks were tested successfully in our experiments. We have excluded the other five benchmarks (apsi, facerec, fma3d, perlbmk and wupwise) that had problems and did not finish the test.

1) Cache Miss Rate Reduction: We first study the cache miss rate reduction with various prefetching algorithms and the AFA prefetcher. Fig. 5 plots the L2 cache miss rate reported by the simulator for the entire twenty-one benchmarks. This series of tests were conducted under eight cases, including the base case (without data prefetching), the cases with individual stream, strided, Markov and MLDT data prefetching, the cases with best-strategy and multi-strategy adaptation and the case with all supported prefetching algorithms running simultaneously.

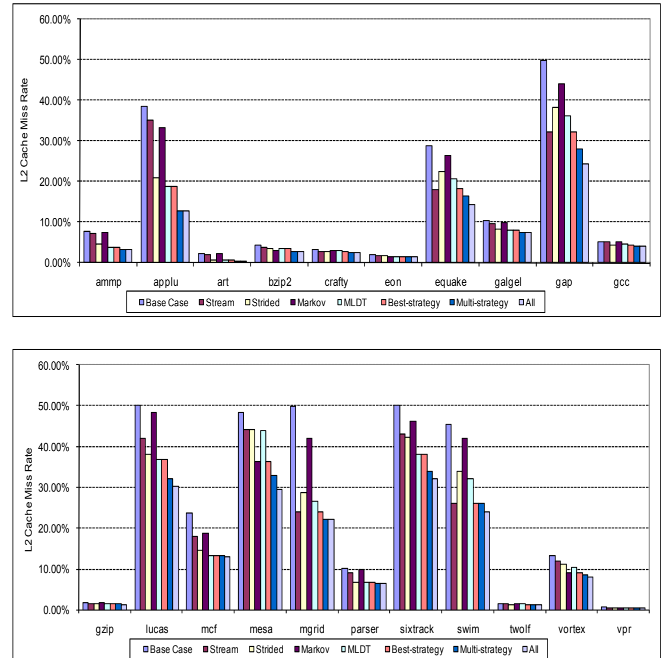


Figure 5. Cache miss rate of SPEC-CPU2000 benchmarks.

As clearly shown from the results, different applications exhibit distinct access patterns, and thus the cache miss rate reduction of various prefetching algorithms have large variations. For instance, stream prefetching significantly reduced the misses for equake, gap, mgrid and swim benchmarks, while not for others like ammp, art, galgel and gcc. Instead, the strided prefetching performed extremely well for ammp, applu, lucas, mcf and etc., and Markov prefetching had considerable miss reduction for bzip2, eon and vortex benchmarks. The MLDT prefetching usually achieved a better miss rate reduction than strided prefetching, but still not for all benchmarks. These observations confirm that an adaptive prefetcher is desired to be able to adjust to different application features at runtime to achieve a better overall prefetching performance. Simply adopting a fixed prefetching strategy cannot be the optimal solution.

The AFA prefetcher has demonstrated its strength through the simulation. From the reported miss rate results, we can tell that the best-strategy can almost achieve the largest miss rate reduction compared with each individual

supported algorithm. This fact confirms that this adaptation strategy can effectively identify the suitable prefetching algorithm for current application access pattern. The multi-strategy adaptation can sometimes achieve an even better miss rate reduction, such as in the applu, gap, lucas, mesa and sixtrack benchmarks. The investigation shows that this further reduction is due to two or three well-performing prefetching strategies the multi-strategy adaptation identified and selected at different stages for these benchmarks. These optimal strategies are all able to generate effective prefetches at a specific stage. Table II lists the primary prefetching algorithms identified and selected by the AFA prefetcher for different benchmarks at runtime.

TABLE II. PRIMARY PREFETCHING ALGORITHMS SELECTED ADAPTIVELY AT RUNTIME

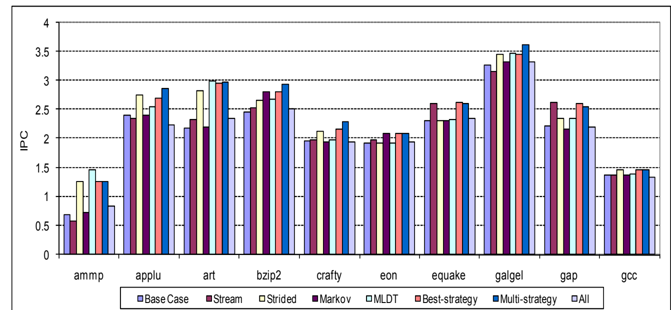
(B-S: Best-strategy, M-S: Multi-strategy; SM: Stream, ST: Strided, MK: Markov, MT: MLDT)

	ammp	applu	art	bzip2	crafty	Eon	equake	galgel	gap	gcc
B-S	*ST*	*ST*	*MT*	*MK*	*ST*	*MK*	*SM*	*MT*	*SM*	*ST*
M-S	*ST*	*ST, MT*	*MT*	*MT, MK*	*ST, SM*	*MK*	*SM*	*ST, MT*	*SM*	*ST*
gzip	lucas	mcf	mesa	mgrid	parser	sixtrack	swim	twolf	vortex	vpr
ST	*ST*	*ST*	*SM*	*SM*	*ST*	*MT*	*SM*	*ST*	*MK*	*ST*
ST, MT	*SM, ST*	*ST*	*SM, MT*	*SM, MT*	*MT, ST*	*ST, MT, MK*	*SM*	*MT, ST*	*MK*	*MT ST*

The last bar within each set of tests represents the miss rate reduction with all four prefetching algorithms working concurrently. The experimental results show that this case has achieved about the same reduction as multi-strategy adaptive prefetching, while better than adaptive strategies sometimes. However, as shown from the reported IPC results presented in the following subsection, the best miss reduction does not translate to best overall performance improvement because this strategy generates extensive replacements to the prefetch destination. In addition, the all-prefetching strategy consumes more resources than an adaptive strategy like the AFA prefetcher because the latter can identify the optimal ones and shut off low-efficiency prefetchers.

2) IPC Improvement: Fig. 6 demonstrates the performance measurement in terms of IPC reported by the SimpleScalar simulator. The results shown in the figure include twenty-one benchmarks under eight cases, similarly as discussed in the previous subsection. The measured IPC results confirm that different prefetching algorithms benefit distinct benchmarks with different patterns. Any specific algorithm did not achieve the best IPC improvement for all benchmarks. Instead, these four supported prefetching algorithms have large variations in terms of the performance gain measured in IPC.

As shown from the reported results, the AFA prefetcher does have the capability to distinguish well-performing algorithms from others and adapts to these selected algorithms to achieve an overall better performance gain. For instance, the best-strategy adaptation has successfully identified strided prefetching suitable for ammp, applu, crafty, gcc, gzip, lucas, mcf, parser, twolf and vpr, while stream prefetching suitable for equake, gap, mesa, mgrid and swim. Both Markov and MLDT prefetching were also identified as better strategies at some cases, like Markov for the benchmark bzip2, eon and vortex, and MLDT for the benchmark art, galgel and sixtrack. Notice that a better cache miss rate reduction does not necessarily result in a better IPC improvement. Take the applu benchmark as an example. The strided prefetching reduced less misses than MLDT did, but it produced better IPC improvement. This is because that MLDT involves more prediction overhead.



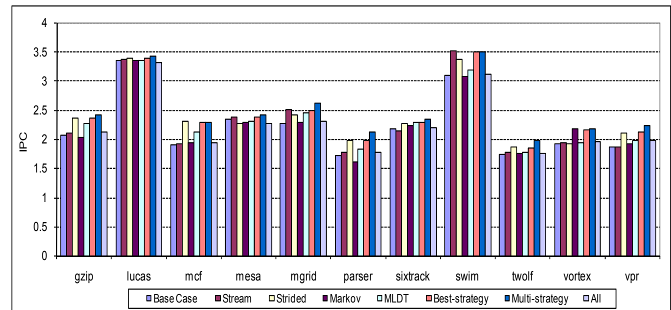


Figure 6. IPC of SPEC-CPU2000 benchmarks with different data prefetching strategies.

It is interesting to notice that multi-strategy adaptation usually generates better performance improvement than best-strategy does. This is because the multi-strategy adaptation is able to recognize multiple well-performing algorithms, and can benefit and complement each other while avoiding low-effective algorithms. Adopting all supported prefetching algorithms does not produce the best performance speedup. This observation reveals that adopting a low-accurate, low-coverage or high-pollution algorithm can even substantially worsen the performance. This fact has also been confirmed from most cases in the experiments.

The average performance improvement of all twenty-one benchmarks with each strategy is shown in Fig. 7. The best-strategy and multi-strategy adaptation mechanisms provided by the AFA prefetcher achieves 15.14% and 17.90%

average improvement respectively, which is clearly better than all other cases. In summary, as verified from the simulation testing, the AFA prefetcher is able to dynamically choose proper algorithms for different applications and to achieve an overall better performance improvement of prefetching.

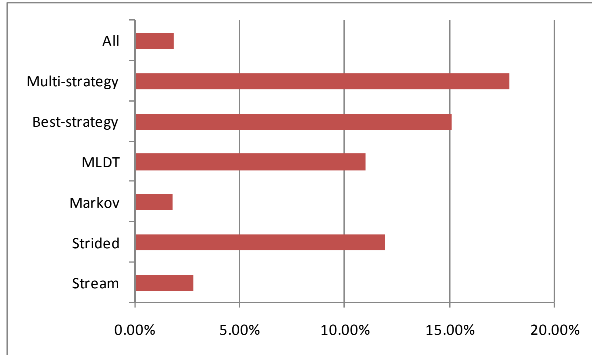


Figure 7. Average IPC speedup

V. Related Work

Data prefetching, as the name indicates, is a technique to fetch data before requested. A similar technique is instruction prefetching, which tries to speculate the future instructions and fetch them from memory in advance [9]. Data prefetching is usually classified as software prefetching and hardware prefetching [28]. Software prefetching is a technique to instrument prefetch instructions to the source code either by a programmer or by a complier during optimization phase. Hardware prefetching does not require modifications to binary or source code, and can benefit directly for existing executables.

The representative hardware prefetching techniques include sequential prefetching, strided prefetching and Markov prefetching. Sequential prefetching [6][7][15] prefetches consecutive cache blocks by taking advantage of locality. One-block-lookahead (OBL) approach automatically prefetches the next block when an access of a block is initiated. However, the limitation of this approach is that the prefetch may not be initiated early enough prior to processor's demand for the data to avoid a processor stall. To solve this issue, a variation of OBL prefetching, which fetches k blocks (called prefetching degree) instead of one block, is proposed. Another variation is called adaptive sequential prefetching, which varies prefetching degree k based on the prefetching efficiency. The prefetching efficiency is a metric defined to characterize a program's spatial locality at runtime. Stream prefetching is a generalized sequential prefetching that supports the detection of multiple streams. [15]. Strided prefetching approach [4][8] observes the pattern among strides of past accesses and thus predicts future references. It builds a state machine to track strides of accesses and generates prefetches when the state machine arrives at a stable state. The strided prefetching is generally implemented with a reference prediction table [4], or the recently proposed Data-Access History Cache [5]. To capture repetitiveness in data reference addresses, Markov prefetching [14] was proposed. This strategy builds a state transition diagram with states denoting an accessed data block. Probability of each state transition is maintained, so that most probable predicted data are prefetched in advance and the least probable predicted data references can be dropped from prefetching.

Other recent efforts in hardware prefetching include Zhou's dual-core execution (DCE) approach [31], Ganusov et al's future execution (FE) approach [10], Sun et al's data push server architecture [26] and Solihin et al.'s memory-side prefetching [24]. DCE and FE approach target for multi-core architecture. They use idle core to pre-execute future loop iterations to warm up cache (bring data to cache in advance). The data push server architecture utilizes a separate processing unit such as a separate core to conduct heuristic prefetching. The memory-side prefetching approach uses a memory processor residing within main memory to observe data access histories and prefetch data proactively upon prediction. It is usually distinguished as push based prefetching from traditional pull based prefetching.

Without the benefit of programmer or compiler hints, the effectiveness of hardware prefetching largely relies on the accuracy of prediction strategies. Incorrect prediction brings useless blocks into cache, consumes memory bandwidth and might cause cache pollution. To increase prefetching accuracy and coverage, hardware prefetching strategies should be able to make dynamic adaptation at runtime for different access patterns. This study targets to provide an algorithm-level hardware adaptive data prefetching to resolve this issue. Some existing literature [6][25] provide various forms of adaptation, however, most of them target at adapting the prefetch degree and prefetch distance only. Our idea is motivated from the fact that no single prediction algorithm can work universally well for all applications. The adaptation at an algorithm-level is a necessity. This study provides such a solution. In addition, the existing studies on adapting prefetch degree and distance are complementary to this study. The Global History Buffer [17] and Instruction-Pointer prefetcher [8] are other hardware prefetching strategies proposed in recent years. However, they both lack adaptation support.

VI. Conclusion

Data prefetching is an effective solution to hiding data-access latency and to mitigating the fast growing processor-memory performance gap. Many hardware prefetching techniques have been widely used in contemporary processor architecture. They are successful for applications with simple data access patterns, but notorious in certain cases for generating pollution and other overhead due to their low effectiveness. Previous study shows this low effectiveness is due to the lack of adaptation of existing hardware prefetchers. This study proposes an Algorithm-

level Feedback-controlled Adaptive (AFA) prefetcher to support algorithm-level adaptation depending on application runtime data access behavior. While existing adaptive prefetchers only adapt within a given prefetching algorithm, AFA can change prefetching algorithms at runtime. It provides more flexibility and, therefore, better performance. We have conducted extensive simulations with an enhanced SimpleScalar simulator to verify and evaluate the design. Simulation results have demonstrated a clear performance improvement over existing strategies.

VII. ACKNOWLEDGEMENT

We are thankful to the anonymous reviewers for their valuable suggestions to further improve this work. This research was supported in part by National Science Foundation under NSF grant CCF-0621435 and CCF-0937877, and by ACM/IEEE High-Performance Computing Ph.D. Fellowship and Illinois Institute of Technology Fieldhouse Research Fellowship.

REFERENCES

[1] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, Vol. 13, No. 7, pp. 422-426, 1970.

[2] D.C. Burger, T.M. Austin and S. Bennett. Evaluating Future Microprocessors: the SimpleScalar Tool Set. University of Wisconsin-Madison Computer Science Department Technical Report 1308, 1996.

[3] S. Byna, Y. Chen and X.-H. Sun. Taxonomy of Data Prefetching for Multicore Processors", Journal of Computer Science and Technology (JCST), vol. 24, no. 3, pp. 405-417, 2009.

[4] T.F. Chen and J.L. Baer. Effective Hardware-Based Data Prefetching for High Performance Processors. IEEE Trans. Computers, pp. 609-623, 1995.

[5] Y. Chen, S. Byna and X.-H. Sun. Data Access History Cache and Associated Data Prefetching Mechanisms. In Proc. of the 2007 ACM/IEEE Supercomputing Conference, 2007.

[6] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors. In Proc. 1993 International Conference on Parallel Processing, pp. I56-I63, 1993.

[7] F. Dahlgren, M. Dubois and P. Stenstrom. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. IEEE Trans. on Parallel and Distributed Systems, Volume 6, Issue 7, pp.733-746, 1995.

[8] J. Doweck. Inside Intel Core Micro-architecture and Smart Memory Access. Intel White Paper, 2006.

[9] A. Falcon, A. Ramirez and M. Valero. Effective Instruction Prefetching via Fetch Prestaging. In Proc. of the 19th International Parallel and Distributed Processing Symposium, 2005.

[10] I. Ganusov and M. Burtscher. Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors. In Proc. of the 14th Annual International Conference on Parallel Architectures and Compilation Techniques, 2005.

[11] B. Goeman, H. Vandierendonck and K. Bosschere. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In Proc. of 7th Intl. Symp. on High performance Computer Architecture, 2001.

[12] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. The 4th edition, Morgan Kaufmann, 2006.

[13] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In Proc. of the 17th International Symposium on Computer Architecture, 1990.

[14] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In Proceedings of the 24th Annual Symposium on Computer Architecture, 1997.

[15] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz and M. T. Vaden. IBM Power6 Microarchitecture. IBM Journal of Research and Development. Vol. 51, No. 6, pp. 639 – 662, 2007.

[16] S.A. McKee. Reflections on the Memory Wall. In Proc. of Computing Frontiers (CF'04), 2004.

[17] K. J. Nesbit and J. E. Smith. Prefetching Using a Global History Buffer. In Proc. of the 10th Annual International Symposium on High Performance Computer Architecture, 2004.

[18] J. Peir, S. Lai, S. Lu, J. Stark and K. Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In Proc. of the 16th International Conference on Supercomputing, 2002.

[19] L. Ramos, J. Briz, P. Ibañez and V. Viñals. Data Prefetching in a Cache Hierarchy with High Bandwidth and Capacity. In ACM Computer Architecture News, pages 37-44, 2007.

[20] E. Rotenberg, S. Bennett, J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, 1996.

[21] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In Proc. of the 30th Intl. Symp. on Microarchitecture, 1997.

[22] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart and X. Zhang. Hardware Counter Driven On-the-Fly Request Signatures. In Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, 2008.

[23] J. Skeppstedt and M. Dubois. Hybrid Compiler/Hardware Prefetching for Multiprocessors Using Low-overhead Cache Miss Traps. In Proc. of the International Conference on Parallel Processing, 1997.

[24] Y. Solihin, J. Lee and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In Proc. of the 8th International Symposium on Computer Architecture, 2002.

[25] S. Srinath, O. Mutlu, H. Kim and Y. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In Proc. of 13th International Symposium on High Performance Computer Architecture, 2007.

[26] X.H. Sun, S. Byna and Y. Chen. Improving Data Access Performance with Server Push Architecture. In Proc. of the NSF Next Generation Software Program Workshop in IPDPS'07, 2007.

[27] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove and M. Hind. Using Hardware Performance Monitors to Understand the Behaviors of Java Applications. In Proc. of the 3rd USENIX Virtual Machine Research and Technology Symp., 2004.

[28] S. P. VanderWiel and D. J. Lilja. When Caches Aren't Enough: Data Prefetching Techniques. IEEE Computer, Vol. 30, No. 7, pp.23-30, 1997.

[29] W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implications of the Obvious. In Computer Architecture News, 23(1):20-24, March 1995.

[30] X. Zhang, S. Dwarkadas, G. Folkmanis and K. Shen. Processor Hardware Counter Statistics As A First-Class System Resource. In Proc. of the 11th Workshop on Hot Topics in Operating Systems, 2007.

[31] H. Zhou. Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window. In Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques, 2005.

[32] Standard Performance Evaluation Corporation, SPEC Benchmarks, http://www.spec.org/