Byna S, Chen Y, Sun XH. Taxonomy of data prefetching for multicore processors. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(3): 405–417 May 2009

Taxonomy of Data Prefetching for Multicore Processors

Surendra Byna, *Member*, *IEEE*, Yong Chen (陈　勇), *Student Member*, *ACM*, *IEEE*
and Xian-He Sun (孙贤和), *Member*, *ACM*, *Senior Member*, *IEEE*

Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois 60616, U.S.A.

E-mail: {sbyna, chenyon1, sun}@iit.edu

Received July 28, 2008; revised December 31, 2008.

Abstract Data prefetching is an effective data access latency hiding technique to mask the CPU stall caused by cache misses and to bridge the performance gap between processor and memory. With hardware and/or software support, data prefetching brings data closer to a processor before it is actually needed. Many prefetching techniques have been developed for single-core processors. Recent developments in processor technology have brought multicore processors into mainstream. While some of the single-core prefetching techniques are directly applicable to multicore processors, numerous novel strategies have been proposed in the past few years to take advantage of multiple cores. This paper aims to provide a comprehensive review of the state-of-the-art prefetching techniques, and proposes a taxonomy that classifies various design concerns in developing a prefetching strategy, especially for multicore processors. We compare various existing methods through analysis as well.

Keywords taxonomy of prefetching strategies, multicore processors, data prefetching, memory hierarchy

1 Introduction

The advances in computing and memory technologies have been unbalanced. Processor performance has been increasing much faster than memory performance over the past three decades. This imbalance has been causing an increasing gap and making memory performance a formidable bottleneck. Since 2004, multicore chips have emerged into mainstream to offer a significant boost in processing capabilities while consuming lower power. Chip multiprocessing (CMP) technology with the help of thread-level parallelism (TLP) and data-level parallelism (DLP) has been the driving processor technology in increasing computing power further. However, the data access problem is getting worse with multiple cores contending for accessing data from memory that is typically shared by these cores.

Data prefetching, which decouples and overlaps data transfer and computation, is widely considered as an effective memory latency hiding technique. Cache misses are a common cause of CPU stalls. Using cache memories effectively enables bridging the performance gap between the processor and memory. To achieve this goal, data prefetching predicts future data accesses of a processor, initiates to fetch data early, and brings the data closer to the processor before the processor requests for the data.

Numerous prefetching strategies have been proposed in the research literature for single-core processors. These strategies predict future data accesses by using recent history of data accesses from which pattern of accesses can be recognized[1−6], by using compiler or user provided hints[7,8], by analyzing traces of past execution of applications or loops[9]. With the emergence of multi-threaded and multicore processors, computing power became abundant. A number of methods have been proposed to utilize this extra computing power for prefetching. Many of these methods run a helper thread ahead of actual execution of an application to predict cache misses[10−15]. Another set of methods employ run-ahead execution at hardware level[16,17], where idle or dedicated cycles are used for prefetching. We proposed to utilize a dedicated server to push data closer to CPU by selecting future data access prediction methods dynamically[18].

Among various strategies mentioned above, questions arise such as what the best prefetching method is to achieve the goal of crossing the data access wall in the multicore era, and what design issues have to be taken into consideration. To address these questions, in this paper, we provide a comprehensive taxonomy of prefetching strategies that primarily captures

Survey

This research was supported in part by the National Science Foundation of USA under Grant Nos. EIA-0224377, CNS-0406328, CNS-0509118, and CCF-0621435.

design issues of prefetching strategies. VanderWiel *et al.*[19] presented a history of prefetching, discussed the general idea of prefetching, and compared various prefetching strategies in the context of single-core processors. Their survey provides a taxonomy addressing what, when, and where (destination of prefetching) questions for hardware prefetching and software prefetching. Oren[20] conducted a survey with a similar classification of hardware and software prefetching methods. With the emergence of multi-thread and multicore architectures, new opportunities and challenges arise in designing prefetching strategies. We propose a complete taxonomy of prefetching mechanisms based on a comprehensive study of hardware and software prefetching, prediction and pre-execution-based prefetching, and more importantly, prefetching strategies that are novel to multicore processors. This taxonomy aims to provide insightful guidelines for making prefetching design and improving performance and productivity of software development.

The rest of the paper is organized as follows. Section 2 presents the taxonomy that classifies data prefetching strategies. In Section 3, we provide a comparison of the pros and cons of existing prefetching methods with examples. We discuss various challenges in implementing prefetching on multicore processors and possible solutions in Section 4 and conclude in Section 5 with a summary.

2 Taxonomy

A data prefetching strategy, whether on single-core or multicore processors, has to consider various issues in order to mask data access latency efficiently. Prefetching strategies should consider both aspects of *what to prefetch* and *when to prefetch*. A strategy should be able to predict future data requirements of an application accurately and to move the predicted data from its source to destination *in time*. Fetching data too early might replace data that would be used by processor in the near future, which causes cache pollution[21]. Fetching data too late wastes bandwidth since a cache miss stall may have already occurred. At the same time, the complexity of executing prefetching methods should be kept low in order not to block the actual processing of an application.

Fig.1 shows three representative scenarios of prefetching strategies. In Scenario A, a prefetch engine (PE) observes history of L1 cache misses and initiates prefetch operations. In multi-threaded or multicore processors, pre-execution-based approaches employ a separate thread to speculate future accesses. In this approach (Scenario B in Fig.1), a compiler or application developer generates computation-thread and prefetching-thread for an application. The prefetching-thread pre-executes slices of code of the main computation-thread and initiates prefetching data into a shared cache memory (L2 cache in Fig.1) earlier than the computation-thread requests. In memory-side prefetching strategy (Scenario C in Fig.1), the prefetching-thread is executed on a memory processor within an intelligent main memory. The predicted data is pushed towards the processor. From these scenarios, it is evident that, in addition to predicting *what* and *when* to prefetch, choosing the source, the destination, and the initiator of prefetching plays a primary role in designing an effective prefetching strategy.

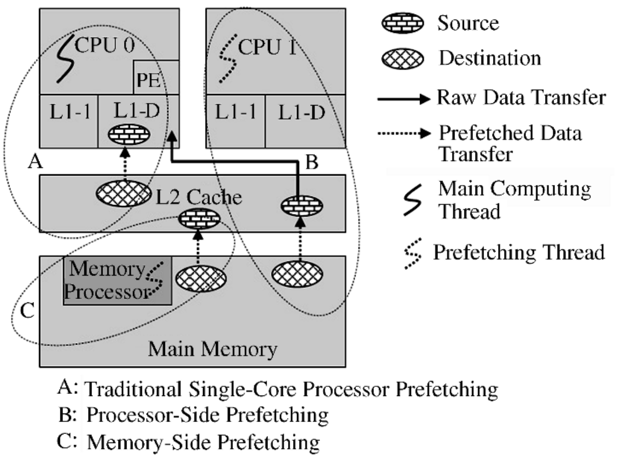


Fig.1. Prefetching scenarios.

We take a top-down approach to characterizing and classifying various design issues, and present a taxonomy of prefetching strategies. Fig.2 shows the top layer of the taxonomy, which consists of the five most fundamental issues that any prefetching strategy has to address: what data to prefetch, when to prefetch, what is the prefetching source, what is the prefetching destination, and who initiates a prefetch. In this section, we examine each element and its taxonomy in detail.

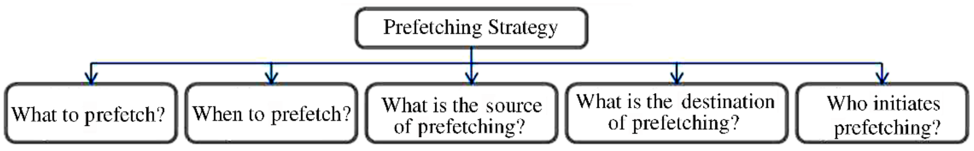

Prefetching Strategy
What to prefetch?
When to prefetch?
What is the source of prefetching?
What is the destination of prefetching?
Who initiates prefetching?

Fig.2. Five fundamental issues in designing a prefetching strategy.

2.1 What to Prefetch?

Predicting what data to prefetch is the most important requirement of prefetching design. In current multi-level memory hierarchies, data that has been recently and frequently used (read/written) by an application is stored at a cache level closer to CPU. When data is not in a cache closer to a processor, a *raw cache miss* occurs, which sends a *demand request* to a lower level cache memory or main memory. Raw cache misses typically cause CPU stalls, and thus the computing power is wasted. If a prefetching strategy can predict the occurrence of such raw misses ahead of time, then a prefetch instruction can be issued early to bring that data by the time it is required to avoid cache misses.

To effectively mask the stall time caused by raw cache misses, the accuracy of predicting *what to prefetch* must be high. Predicting future data references accurately is critical to a data prefetching strategy. If the prediction accuracy is low, useless data blocks are fetched into the upper levels of cache, which might replace data blocks that would be used in the near future. This mis-prediction leads to *cache pollution*, which in turn causes poor cache performance and overall performance degradation. Intuitively, data prefetching is effective when application requests follow regular patterns. Execution of code in loops is usually a target by various prefetching strategies, where regular data access patterns are common.

Fig.3 shows further classification of various methods that are used in predicting *what* data to prefetch. *Hardware-controlled strategies* predict future accesses using history or run-ahead execution or offline analysis. *Software-controlled strategies* utilize compiler or user inserted prefetching instructions, or post-execution analysis. Hybrid-controlled strategies also use history-based approaches or pre-execute slices of code.

2.1.1 Hardware-Controlled Strategies

In a *hardware-controlled* strategy, prefetching is entirely managed by hardware. Various methods support hardware-controlled prefetching. *Online history-based prediction* approach observes history of accesses and analyzes them to find regular patterns among the accesses. Instead of relying on history of data accesses, *run-ahead execution*[22,23] approach pre-executes future instructions while data cache misses are outstanding. *Offline analysis* uses history of previous execution of an application in prefetching for a future execution.

Online history-based prediction is the most commonly used hardware controlled data prefetching strategy. In this strategy, a *prefetch engine* (*PE*) predicts future data references and issues prefetching instructions. The prefetching logic is completely implemented within a processor, and this strategy does not require any user interference. PE observes either the history of data accesses or the history of cache misses to predict future accesses. For instance, Intel Core microarchitecture uses a Smart Memory Access[24] approach, where an instruction-pointer-based prefetcher tags the history of each load instruction, and if a constant stride is found, the next address is calculated. Data at the calculated address is prefetched into L1 cache. Numerous prediction algorithms have been proposed to find patterns among history of accesses or cache misses. We elaborate prediction algorithms and data access patterns in Subsection 2.1.4 that all history-based prediction strategies try to predict. Online history-based analysis is beneficial to applications with regular data access patterns. If there are no regular patterns, the overhead in predicting future accesses may not be beneficial. In some cases, with the added cost in finding patterns, there may be no gain, especially when data accesses are completely random.

Runahead execution exploits idle cycles or cores to run instructions speculatively. The main idea behind this approach is to utilize the power of multicore processors, when they are not busy. Zhou[17] and Ganusov *et al.*[16] proposed to utilize idle cores of a Chip Multiprocessor (CMP) to speed up single threaded programs.

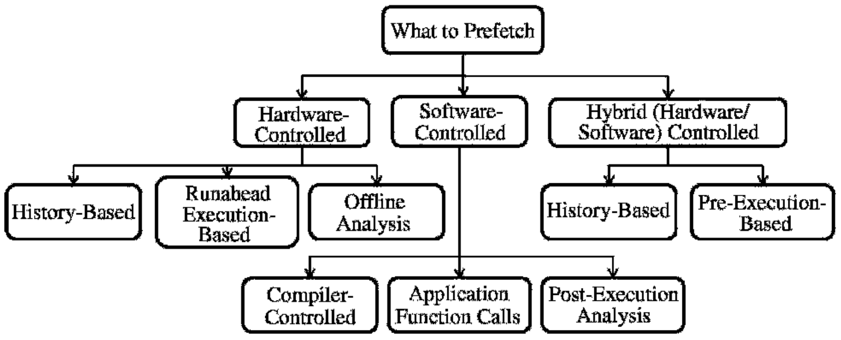


Fig.3. Predicting *what* data to prefetch.

Zhou's *dual-core execution* (DCE) approach takes advantage of idle cores to construct a large, distributed instruction window, and Ganusov *et al.*'s *future execution* (FE) approach uses an idle core to pre-execute future loop iterations using value prediction.

Runahead execution is beneficial to applications with regular or random accesses. Exploiting idle cycles of unused resources in processors improves their utilization and application performance. However, runahead execution requires special hardware implementation to pre-execute instructions. Also, the dependence of these methods on availability of idle cycles may be a hurdle in broadly applying these methods. This problem can be solved with dedicated hardware to provide prefetching support.

Offline analysis strategy is another hardware-controlled prefetching approach. Kim *et al.*[9] proposed such a method to analyze data access patterns for hotspots of code that are frequently executed. After a hotspot executes for the first time, its data accesses are analyzed and the result, the pattern information, is stored. This pattern information is used for future runs of that hotspot. This approach works well for applications that refer to similar data access patterns or that call a function repeatedly. Obtaining address traces needs availability of special hardware or profiling in application.

While hardware-controlled strategies are widely used, a significant drawback is that software developers have limited control over prefetching, typically to turn prefetching on and off. In addition, poor prediction accuracy of hardware PE may result in cache pollution and limited performance speedup.

2.1.2 Software-Controlled Strategies

Software-controlled prefetching[25,26] strategies enable a programmer or a compiler to insert prefetch instructions into programs. The motivation behind these strategies is the higher possibility of a compiler or developers having better knowledge of the application's data requirements, which makes it promising to gain more from software prefetching. Software-controlled prefetching can use compiler-controlled prefetch instructions, or inserting prefetching function calls in the source code or inserting prefetching instructions based on post-execution analysis. Many processors provide support for such prefetch instructions in their instruction set. Compilers or developers can insert prefetch instructions or routines provided by compilers (e.g., *__builtin_prefetch*() in *gcc* and *sparc_prefetch_read_once*() in Sun cc on SPARC processors). *Post-execution analysis* can also be used as software-controlled prefetching approach, where traces of data accesses are analyzed offline for finding patterns. This pattern information is used to prefetch data at runtime.

A considerable disadvantage of software-controlled prefetching is that it imposes a heavy burden on developers and compilers, and is less effective in overlapping memory access stall time on ILP (Instruction Level Parallelism)-based processors due to potential late prefetches and resource contention[13]. Having an automated toolkit or an advanced compiler optimization for converting the knowledge of pattern analysis into prefetching function calls reduces burden on developers.

2.1.3 Hybrid Hardware/Software-Controlled Strategies

Hybrid hardware/software-controlled strategies are gaining popularity on processors with multi-thread support. On these processors, threads can be used to run complex algorithms to predict future accesses. These methods require hardware support to run threads that are specifically executed to prefetch data. They also require software support to synchronize the prefetching thread with the actual computation thread. The hybrid hardware/software-controlled prefetching strategies can be further categorized into methods that analyze history of data accesses of computation threads and that pre-execute data intensive parts of the computation thread to warm up a shared cache memory by the time raw cache misses occur.

History-based hybrid prediction strategies usually employ a hardware-supported multi-threading mechanism to analyze history of accesses to predict future accesses, and to prefetch data. For instance, Solihin *et al.*[13] proposed memory-side prefetching, where an intelligent memory processor resides within the main memory, and a thread running on the memory processor analyzes data access history of data accesses to predict future references. This scheme observes stride-based and pair-based correlations among past L2 cache misses and pushes predicted data to L2 cache. Similar to hardware-controlled history-based prefetching methods, history-based hybrid strategies are not highly beneficial with random access patterns.

Pre-execution-based methods use a thread to execute slices of code ahead of main computation thread. Many such prefetching strategies have been proposed to utilize hardware-supported multithreading. A small list of various proposals includes Luk *et al.*'s software-controlled pre-execution[27], Liao *et al.*'s software-based speculative pre-computation[28], Zilles *et al.*'s speculative slices[15], Roth *et al.*'s data-driven multithreading[29], Annavaram *et al.*'s data graph pre-computation[10], and Hassanein *et al.*'s data forwarding[30]. Many of these methods rely on compiler support to select slices of code to pre-execute and to trigger execution of speculative code. In contrast, Collins *et al.*[31,32] suggest using hardware to select instructions for pre-computation.

Pre-execution-based methods are useful in predicting regular and random accesses. However, developers or compiler have to separate pre-execution threads in order to run ahead. Synchronization to run pre-execution thread early enough to prefetch is a challenging task as well.

2.1.4 *Classification of Data Access Patterns*

Hardware-controlled, software-controlled, and hybrid hardware/software-controlled approaches largely use prediction algorithms based on history of data accesses or cache misses. These prediction algorithms search for regular patterns among history of data accesses. Fig.4 shows a classification of data access patterns based on spatial distance between accesses, the repeating behavior, and the request size. Spatial patterns are further divided, based on the number of bytes (also called *strides*) between successive accesses, as contiguous, non-contiguous, and combinations of both. Non-contiguous patterns can be further classified by the property of strides between accesses. These data access patterns may occur multiple times when loops or functions are executed repeatedly. We classify these patterns as either single occurrence or repeating patterns. The request size of accesses in each pattern may be fixed or variable. This classification captures a wide range of data accesses.

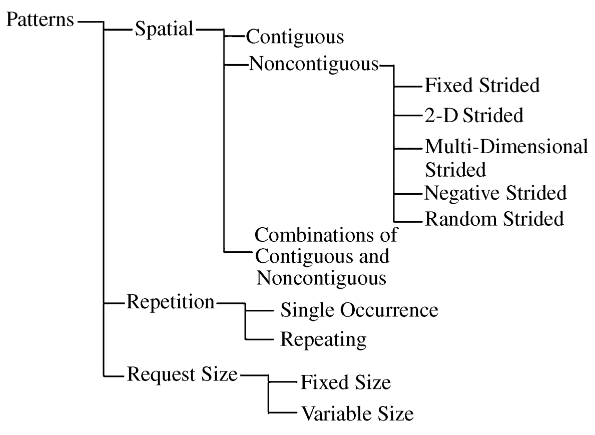


Fig.4. Classification of data access patterns.

Several prediction algorithms have been proposed to find various patterns that are shown in Fig.4. Sequential prefetching[3,4] fetches consecutive cache blocks by taking advantage of locality. One-block-look-ahead (OBL) approach automatically prefetches the next block when an access of a block is initiated. However, the drawback of OBL prefetching 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, was proposed. Another variation, called adaptive sequential prefetching, 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. Stride prefetching approach[2,3,24] predicts future accesses based on strides of the recent history. Various strategies have been proposed based on stride prefetching, and these strategies maintain a Reference Prediction Table (RPT) to keep track of recent data accesses. RPT acts like a separate cache and holds data access information of the recent memory instructions. RPT provides an effective method to implement stride prefetching, but it can only capture constant strides. To capture repetitiveness of data accesses, Markov prefetching[5] was proposed. This strategy assumes that history might repeat itself among data accesses and builds a state transition diagram with states denoting accessed data blocks. The probability of each state transition is maintained, and data accesses repeating with high probability are selected as prefetching candidates. The k-th *order* Markov predictor uses the last k requests from the sequence to make predictions of the next data accesses. Distance prefetching[6] uses Markov chains to build and maintain probability transition diagram of strides (or distances) among data accesses. Multi-Level Difference Table (MLDT)[33] uses time-series analysis method to predict future accesses in a sequence, by finding the differences in a sequence to multiple levels. Nesbit *et al.*[34] proposed a Global History Buffer in order to combine multiple prediction algorithms. Chen *et al.*[35] suggested a buffer called Data Access History Cache (DAHC) to enable multiple history-based prediction algorithms to find patterns among applications' memory accesses.

2.2 When to Prefetch?

The timing to issue a prefetch instruction has significant effect on the overall performance of prefetching. Prefetched data should arrive at its destination before a raw cache miss occurs. The efficiency of timely prefetching depends on total prefetching overhead (i.e., the overhead of predicting future accesses plus the overhead in moving data) and the time window for the occurence of next cache miss. If the total prefetching overhead exceeds the time window, adjusting prefetching distance can avoid late prefetches. Fig.5 shows a classification of various methods used in deciding when to prefetch, namely event-based, lookahead program-counter-based, software-controlled synchronization, and prediction-based.

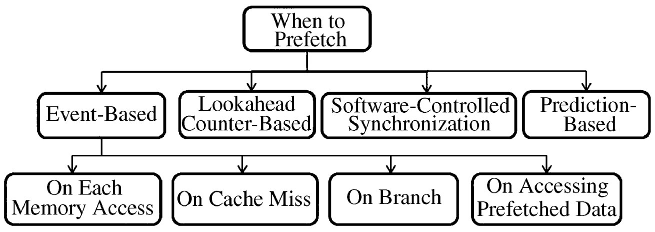


Fig.5. Methods of deciding *when* to prefetch.

Event-based mechanism issues a prefetch instruction upon occurrence of some event, such as a memory reference, or a cache miss, or a branch, or accessing a previously prefetched data block for the first time. Prefetching on each memory reference is also called *Always prefetch*. The prefetching decision is straightforward, however, the possibility of useless prefetches is high for this approach. Prefetch on a miss is a common implementation on existing processors as it is simple to implement. Tagged prefetching[36] initiates a prefetch instruction when a data access hits previously prefetched data block for the first time. Branch-directed prefetching[36] suggests that, since branch instructions determine which instruction path is followed, data access patterns are also dependent upon branch instructions.

Chen *et al.*[2] proposed using a *lookahead program counter* (LA-PC) to decide on when to initiate prefetches. In loop codes, hiding the memory latency by prefetching depends on the execution time of one loop iteration. If the loop execution time is too little, the prefetching overhead may be higher. To solve this problem, instead of prefetching one iteration ahead, the lookahead prediction adjusts prefetching distance using a pseudo counter, called LA-PC that remains a few cycles ahead of actual PC.

Software-controlled prefetching approaches require either compiler or application developers to make decision to insert prefetch functions to prefetch data early enough. Mowry *et al.*[26] provide an algorithm to calculate the prefetching distance[25]. According to this algorithm, prefetching instructions are called strictly for the data references that would cause cache misses. The innermost loop is unrolled for all the references that do not cause a cache miss, i.e., the degree of loop unrolling is equal to the cache block reuse. This algorithm avoids unnecessary prefetch instructions and reduces the overhead. The number of loop iterations needed to fully overlap a prefetching access is called the prefetching distance. Assuming memory access latency is l, and the work per loop iteration is w, the right prefetch distance can be calculated as $\lceil l/w \rceil$. An epilogue loop is called without prefetching to execute the last few iterations that do not fit in the main loop. In helper-thread-based approaches, periodic synchronization of computation thread with helper-thread is required to prevent late prefetches or too early prefetches. Compilers or application developers define how earlier the prefetching thread should run than the computation thread to initiate prefetching. A sample-based or dynamic triggering mechanism controls a helper-thread to execute a limited number of iterations ahead of the computation thread. This synchronization mechanism also targets at preventing helper-thread execution lagging behind the computation thread[12,14].

In many applications, data access bursts follow certain patterns. By analyzing the time intervals, future data bursts can be predicted, and decided when to start prefetching. *Prediction-based* decision of when to prefetch has been applied in I/O prefetching[37], but has not been researched much for memory level prefetching due to the cost of prediction. Server-based push prefetching[38] proposed using prediction of when to prefetch since the cost of prediction is moved to a dedicated server.

2.3 Source of Prefetching

Memory hierarchy contains multiple levels including cache memories, main memory, secondary storage, and tertiary storage. Data prefetching can be implemented at various levels of memory hierarchy (Fig.6). Data can be prefetched between cache memories and main memory, or between main memory and storage. To design a prefetching strategy, it is necessary to consider where the latest copy of data is. In existing deep memory hierarchies with write-back policy, data can reside

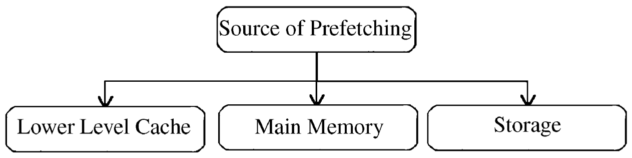


Fig.6. Source of prefetching.

at any level of memory hierarchy. In single-core processors, prefetching source is usually the main memory or a lower level cache memory. In multi-core processors, memory hierarchy contains local cache memories that are private to each core and cache memories that are shared by multiple cores. Designing a prefetching strategy considering multiple copies of a data in local cache memories may lead to data coherence concerns, which is a challenging task. When data is shared, finding the source with the latest copy of data is necessary.

2.4 Destination of Prefetching

Destination of prefetching should be designed more carefully to deal with cache thrashing and cache congruence[39]. Prefetching destination should be closer to CPU than to a prefetching source in order to obtain performance benefits. As shown in Fig.7, data can be prefetched either into a cache memory that is local to a processor, or into a cache memory that is shared by multiple processing cores, or to a separate prefetch cache. A separate prefetch cache can be either private to a core or shared by multiple cores.

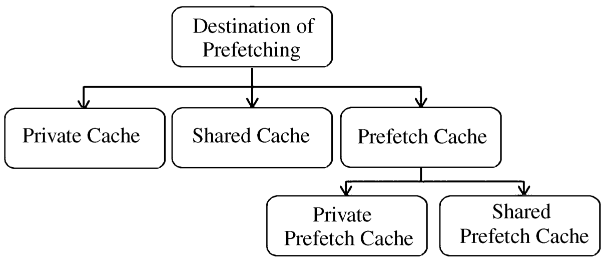


Fig.7. Destination of prefetching.

While the best destination of prefetching is the cache level closet to the processor, there are various issues that affect such prefetch decision. One of them is the limited size of the cache memory. Prefetching data into the top level cache hierarchy may have more impact on polluting the cache and replacing useful cache lines. Cache thrashing is a problem for cache memories that have low associativity. Multiple blocks of data try to occupy the same cache lines causing eviction of previously occupied data blocks from the cache that is being actively used. Prefetching can increase the severity of cache thrashing, where prefetched data replaces active cache lines. Improving prefetching accuracy can reduce replacing useful cache lines and effectively decrease the possibility of cache thrashing. A dedicated buffer called prefetch cache[40] was thus proposed to achieve this goal. In multicore processors, prefetching destination varies. Each core may prefetch data to its private cache or its private prefetch cache. Another scenario is that one of the cores prefetches data into a shared cache[11] (e.g., helper-thread based preexecution). A prefetching strategy should consider the destination of the prefetching carefully in order to minimize the effect of cache pollution and to maintain coherence of prefetched data. On the other hand, a provision of prefetch cache requires modification to conventional memory hierarchy to lookup in this cache as well before proceeding to lookup in the next level of the hierarchy.

Replacement algorithms in selecting victim lines in a cache congruence class (set) should also be designed carefully. When prefetched lines are placed in the regular cache, high frequency of prefetching can increase replacing useful cache lines. Casmira *et al.*[21] proposed Prefetch Buffer Filter (PBF), a small fully associative buffer, to reduce the effect of cache pollution. This buffer holds a prefetched cache line in PBF until it is accessed for the first time. Then, the prefetched cache line is moved into cache. Jain *et al.*[41] suggested using software instructions to augment LRU replacement policy. These instructions allow a program to evict a cache element by making it the least recently used element or to keep a cache element in the cache. Replacement policies that select victims based on both frequency and recency of accesses, such as Adaptive Replacement Cache (ARC)[42] can also reduce cache pollution.

2.5 Who Initiates Prefetch Instructions?

Prefetch instructions can be issued either by a processor that requires data or by a processor that provides such a prefetching service. The first method is generally called *client-initiated* or *pull-based* prefetching, while the latter is called *push-based* prefetching. Fig.8 shows a further classification of *pull-based* and *push-based* strategies depending on where the initiator is located.

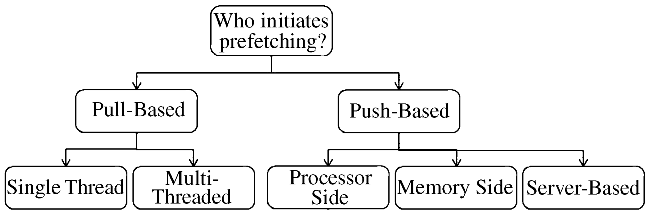


Fig.8. Initiator of prefetching.

Pull-based prefetching has been a common approach of prefetching in single-core processors. In this method, prefetching logic (prediction and initiation) resides within the processor. *Multi-threaded* processors enable

Table 1. Summary of the Pros and Cons of Prefetching Strategies

Prefetching Method	Pros	Cons	Examples*
Hardware-Controlled Prefetching	• No need of user intervention • Good for applications with simple strided patterns	• Generates more unnecessary prefetches than software controlled prefetching[20] • Needs special hardware support	One-block lookahead, sequential prefetching[3,4], strided prefetching[2], Markov prefetching[5], distance prefetching[6], multi-level difference table[33], off-line training of Markovian predictors[9], dual-core execution[17], future execution[16], run-ahead execution[22]
History-Based Prediction	• Useful in hardware-controlled, software-controlled and in hybrid-controlled strategies • Good for applications with regular patterns	• Prediction complexity high for complex patterns • Useless for random data accesses	One-block lookahead, sequential prefetching[3,4], strided prefetching[2], Markov prefetching[5], distance prefetching[6], multi-level difference table[33], user-level memory thread[13], IO prefetching with signatures[1]
Run-Ahead Execution Based	• No need of history of accesses • Uses idle cycles of single-core processors or a separate core of multi-core processors • Good for applications with regular or irregular patterns	• Special hardware to pre-execute instruction is needed	Dual-core execution[17], future execution[16], run-ahead execution[22]
Offline Analysis	• Good for applications with repeating function calls with loops or for those with same access patterns repeating in each run of an application	• Hard to capture variable patterns and pointer references. • May need special hardware or a tool to modify source code at software level after offline analysis to include prefetching instructions	Off-line training of Markovian predictors[9]
Software-Controlled Prefetching	• Better accuracy of what to prefetch • Good for applications with loop code, when there is enough computation to overlap with prefetching data	• Compiler or application developer has to insert prefetching functions in source code • Prefetching distance must be adjusted and loops have to be unrolled accordingly	Software-controlled prefetching in shared-memory multiprocessors[26], software-controlled pre-execution[27], software-based speculative pre-computation[28]
Pre-Execution-Based Prefetching	• Can predict data accesses by running ahead of computation thread • Promising on multi-core processors • Good for applications with regular or irregular patterns	• Compiler or application developer has to create pre-execution thread • Synchronization of helper-thread and computation thread is challenging	Software-controlled pre-execution[27], software-based speculative pre-computation, speculative slices[15], data-driven multithreading[29], data graph pre-computation[1], data forwarding[30], IO prefetching with pre-execution[11]
Pull-Based Prefetching	• Easier to monitor cache misses on processor side • Pre-execution on multi-core processors show promise • Good for applications with simple regular patterns	• Predicting complex patterns or running pre-execution threads may compete for cycles with original computing	One-block lookahead, sequential prefetching[3,4], strided prefetching[2], Markov prefetching[5], distance prefetching[6], software-controlled pre-execution[27], software-based speculative precomputation[28], speculative slices[15], data-driven multithreading[29], dual-core execution[17], future execution[16]
Push-Based Prefetching	• Data transfer can be decoupled from computation effectively • Possible to run aggressive prediction algorithms • Good for applications with complex regular patterns or irregular patterns	• Needs special hardware to monitor data accesses at processor cores • May become a bottleneck if too many processor cores request service from the same memory processor	User-level memory thread[13], multi-level difference table[38], push prefetching for pointer-intensive applications[43], data forwarding[30]

*These examples are only representative proposed prefetching methods in an exhaustive list.

decoupling of data access from computing. Helper-thread-based prefetching[27,29] is a representative strategy that pulls data closer to a processor from main memory.

In *push-based* prefetching, a core other than the actual computation core fetches data. Run-ahead execution[16,17] strategies are such examples. Helper-thread-based prefetching[27,29] can also be placed on a separate core on *processor side* to push data into a shared cache that is used by the computation core as well.

Memory-side prefetching is relatively a new idea, where a processor residing in the main memory pushes predicted data closer to the processor[43]. *Server-based* strategy pushes data from its source to destination without waiting for requests from processor side. Data Push Server (DPS)[33] uses a dedicated server to initiate and proactively pushes data closer to the client *in time*.

Both pull-based and push-based methods have the pros and cons. The efficiency of pull-based prefetching is largely limited by the complexity of prediction algorithms. In pre-execution-based prefetching, with the use of helper-threads, synchronization is needed to initiate pre-execution. Intuitively, with the assumption of the same prediction overhead and same accuracy as those of client-initiated prefetching, push-based prefetching with a dedicated hardware support is better than pull-based prefetching methods since push-based prefetching moves the complexity of prefetching outside the processor. Another benefit of push-based prefetching is that it can be faster as main memory does not have to wait for a prefetching request from the processor. However, the scalability of the memory processor can become an issue when a large number of processing cores have to be served in memory-side prefetching. *Server-based* push prefetching solves this problem by using dedicated server cores.

2.6 Summary

Table 1 provides a summary of broad categories of various prefetching strategies and their pros and cons with examples of various published literature.

3 Comparison of Existing Prefetching Mechanisms

Table 2 presents a detailed comparison of selected prefetching strategies that are published in research literature and their categorization based on the taxonomy we present in the previous section. While there are many other published researches in prefetching, the selected set of strategies for comparison is representative of others.

The first four strategies shown in the table were originally designed for single-core processors. However, the prediction methods of these strategies can be used in identifying future accesses of multicore processors as well. In addition, they were proposed for hardware-controlled prefetching, but their prediction algorithms can be used for software-controlled prefetching as well. Kandiraju *et al.*[6] proposed their method for translation look-aside buffer (TLB), but their prediction method can also be applied to regular caches.

There are many processor-side initiated prefetching methods. Among them, dependence graph generator-based prefetching method is a hardware-controlled strategy that scans the pre-decoded instructions and load/store instructions that are deemed likely to cause cache misses marked for prefetching in [1]. Zhou[17] and Ganusov *et al.*[16] use idle cycles of a core in a dual-core processor to perform prefetching for the other core. Prefetching is initiated on the processor side and data is fetched into a shared L2 cache from main memory.

Among memory-side initiated prefetching strategies, Solihin *et al.*[13] use a helper-thread on the memory-side to push data into L2 cache. The prefetching method proposed by Luk *et al.*[27] is also a memory-side initiated and helper-thread-based approach, but uses software-controlled synchronization for the computation thread and the pre-execution thread. Hardware support can improve the efficiency of the synchronization of these threads even further. Speculative Slices method is a memory-side initiated approach as well, but uses hybrid (hardware/software)-controlled helper-threads. Data forwarding[30] is a hybrid-controlled memory-side prefetching approach that pushes data from main memory to L1 cache and registers with software-based synchronization.

Sun *et al.* proposed a prefetching method that utilizes a dedicated core for prefetching in a multicore processor[18]. This core is the prefetching server for other client computing nodes. It employs a hardware-controlled prefetching engine and multi-level difference-table-based prediction to identify future accesses, which can be implemented at software level as well with a thread. This method can be implemented at lower level cache or at main memory level and prefetches data into a special private prefetch cache. This method is a push-based prefetching strategy, where the server proactively pushes data into the private prefetch cache.

Table 2. Comparison of Prefetching Strategies Based on the Taxonomy

Publication	What?	When?	Source	Destination	Initiator
Dahlgren *et al.*[3,4]	Hardware-controlled/software-controlled, next k blocks, (sequential prefetching)	Event-based	Lower level cache/ main memory	Private, L1 cache/ L2 cache	Processor side
Chen *et al.*[2]	Hardware-controlled/software-controlled, constant regular strides, (strided prefetching)	Event-based/lookahead counter-based	Lower level cache/ main memory	Private, L1 cache/ L2 cache	Processor side/ memory side
Joseph *et al.*[5]	Hardware-controlled/software-controlled, repeating data accesses, (Markov chain based prediction)	Event-based	Lower level cache/ main memory	Private, L1 cache/ L2 cache	Processor side/ memory side
Kandiraju *et al.*[6]	Hardware-controlled/Software-controlled, repeating strides, (Markov chain based distance prediction)	Event-based	Lower level cache/ main memory	Private TLB, Private L1 cache/ L2 cache	Processor side/ memory side
Annavaram *et al.*[10]	Hardware, precomputation-based (data graph pre-computation)	Event-based	Main memory	L1/L2 cache	Processor side
Ganusov *et al.*[16]	Hardware-controlled, run-ahead execution-based	Event-based	Main memory	Shared L2 cache	Processor side
Kim *et al.*[9]	Hardware-controlled, offline analysis	Event-based	L2 cache/main memory	L1 cache/L2 cache	Processor side
Luk *et al.*[27]	Software-controlled, helper-thread	Software-controlled synchronization	Main memory	L1/L2 cache	Memory side
Zilles *et al.*[15]	Hybrid-controlled, helper thread-based	Software-controlled synchronization	Main memory	L1/L2 cache	Memory side
Solihin *et al.*[13]	Hybrid-controlled, history-based prediction for pair-wise correlation	Event-based	Main memory	L2 cache	Memory side
Hassanein *et al.*[30]	Hybrid-control, helper-thread-based	Software-controlled synchronization	Main memory	Private, L1 cache and CPU registers	Memory side
Byna *et al.*[33]	Hardware-controlled/software-controlled, complex and nested regular patterns, (multi level difference table-based prediction)	Prediction-based	Lower level cache/ main memory	Private prefetch cache	Memory side

4 Challenges in Prefetching for Multicore Processors

In addition to the design considerations mentioned in Section 2, prefetching strategies for multicore processors include more challenges. These challenges include resolving multiple computing cores' competition for memory bandwidth, maintaining coherency of prefetched data, and balancing usage of idle cycles for prefetching vs. using them to do extra computing.

Resolving potential competition for memory bandwidth from multiple cores is a challenging task and a highly probable performance bottleneck. In single-core processors, main memory accepts prefetching requests for only one core. In multicore processors without prefetching, data access requests from multiple cores can potentially cause severe contention at shared cache memory or main memory, when too many requests overwhelm the bandwidth of that level of memory hierarchy. If prefetching is not performed properly, prefetching requests from multiple cores may impose even more pressure on main memory. For example, the memory-processor-based solutions[13,30] are not scalable to monitor data access history or to pre-execute threads and predict future references for multiple cores. One way to solve this problem is to decouple data prefetching accesses from raw cache misses from computing cores. In addition, prefetching accuracy has to be high to avoid useless prefetching requests. A high accuracy of predicting future accesses can be achieved with dynamic selection of prediction algorithms based on different data access patterns. Moreover, prefetch requests have to be scheduled in a way to avoid competition.

Another challenge of multicore processor prefetching is maintaining cache coherence. Multicore processors access the main memory, which is shared by multiple cores, and hence, at some level in the memory hierarchy (multiple levels of cache memories), they have to resolve conflicting accesses to memory. Cache coherence in multicore processors is typically dealt with either by using directory-based approach or by using snooping cache accesses. With prefetching, the probability of having stale copies of data is higher if prefetching is performed too early and other cores are modifying that data. Prefetching in a timely manner reduces the risk to some extent. The coherence problem can also be solved by looking into directory and dropping prefetching requests if a data block is shared by multiple cores. If a data block is modified by another core after it is prefetched, then the prefetched block has to be invalidated or updated to maintain coherence.

Usage of aggressive prediction algorithms on single-core processors has long been discouraged as their complexity may become counter-productive. With large amount of computing capability available on multicore processors, complex prediction algorithms can be run to identify future data accesses. However, there should be a balance between performance gains obtained with prefetching by using computational resources, and the performance that would have been obtained if those resources were spent on doing actual computation.

One may argue that more computation might have been finished if the resources were used to do actual computation. In the era of multicore processors and being able to keep billions of transistors on single chip, special hardware cores, whose purpose is prefetching for other cores, can be implemented. It is time to use complex prediction algorithms by transferring their complexity to dedicated cores as we proposed in the server-based push prefetching architecture[18]. We proposed to use a dedicated server core to provide data access support by predicting and prefetching data for computing cores. This server core adaptively chooses prediction and scheduling strategies based on data access patterns and supports data access for multiple cores.

5 Conclusions

The performance gain of a prefetching strategy depends on various criteria. With the emergence of multicore and multi-threaded processors, new challenges and issues need to be considered in designing and developing an effective prefetching strategy. In this paper, we provide a comprehensive taxonomy of data prefetching strategies based on the five fundamental issues (*what, when, destination, source, and initiator*) of a prefetching strategy design. We discuss each of these issues and how they impact the design of a prefetching strategy using a systematic study of various existing strategies. Based on the taxonomy, we compare a set of representative existing prefetching strategies.

We also discuss challenges of prefetching strategies in multicore processors and present potential solution in this study. In addition to the five fundamental issues, prefetching in multicore processors should also consider maintaining cache coherence, reducing the amount of bandwidth contention due to prefetching requests, and utilizing extra computing power offered by multicore processors for running complex prediction algorithms. A prefetching strategy for multicore processing environments has to be adaptive to choose among multiple methods to predict future data accesses. When a data access pattern is easy to be found, prefetching strategy can choose history-based prediction algorithms to predict future data accesses. If data accesses are random, using pre-execution-based approach would be beneficial. In server-based push prefetching, we base our prefetching strategies considering these challenges.

References

[1] Byna S, Chen Y, Sun X-H, Thakur R, Gropp W. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proc. the International Conference on High Performance Computing, Networking, Storage and Analysis* (*SC'08*), Austin, USA, November 2008, Article No. 44.

[2] Chen T F, Baer J L. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 1995, 44(5): 609–623.

[3] Dahlgren F, Dubois M, Stenström P. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proc. International Conference on Parallel Processing*, New York, USA, Aug. 16–20, 1993, pp.56–63.

[4] Dahlgren F, Dubois M, Stenström P. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, July 1995, 6(7): 733–746.

[5] Joseph D, Grunwald D. Prefetching using Markov predictors. In *Proc. the 24th International Symposium on Computer Architecture*, Denver, USA, June 2–4, 1997, pp.252–263.

[6] Kandiraju G, Sivasubramaniam A. Going the distance for TLB prefetching: An application-driven study. In *Proc. the 29th International Symposium on Computer Architecture*, Anchorage, USA, May 25–29, 2002, pp.195–206.

[7] Luk C K, Mowry T C. Compiler-based prefetching for recursive data structures. In *Proc. the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, USA, Oct. 1–5, 1996, pp.222–233.

[8] Rabbah R M, Sandanagobalane H, Ekpanyapong M, Wong W F. Compiler orchestrated pre-fetching via speculation and predication. In *Proc. the 11th International Conference on Architecture Support of Programming Languages and Operating Systems*, Boston, USA, Oct. 7–13, 2004, pp.189–198.

[9] Kim J, Palem K V, Wong W F. A framework for data prefetching using off-line training of Markovian predictors. In *Proc. the 2002 IEEE International Conference on Computer Design*, Freiburg, Germany, Sept. 16–18, 2002, pp.340–347.

[10] Annavaram M, Patel J M, Davidson E S. Data prefetching by dependence graph precomputation. In *Proc. the*

28th International Symposium on Computer Architecture, Göteborg, Sweden, June 30–July 4, 2001, 29(2): 52–61.

[11] Chen Y, Byna S, Sun X-H, Thakur R, Gropp W. Hiding I/O latency with pre-execution prefetching for parallel applications. In *Proc. the International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC'08*), Austin, USA, November 2008, Article No.40.

[12] Kim D, Liao S S, Wang P H, del Cuvillo J, Tian X, Zou X, Wang H, Yeung D, Girkar M, Shen J P. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In *Proc. the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, Palo Alto, USA, March 21–24, 2004, p.27.

[13] Solihin Y, Lee J, Torrellas J. Using a user-level memory thread for correlation prefetching. In *Proc. the 29th International Symposium on Computer Architecture*, Anchorage, USA, May 25–29, 2002, pp.171–182.

[14] Song Y, Kalogeropulos S, Tirumalai P. Design and implementation of a compiler framework for helper threading on multicore processors. In *Proc. the 14th Parallel Architectures and Compilation Techniques*, St. Louis, USA, Sept. 17–21, 2005, pp.99–109.

[15] Zilles C, Sohi G. Execution-based prediction using speculative slices. In *Proc. the 28th International Symposium on Computer Architecture*, Göteborg, Sweden, June 30–July 4, 29(2): 2–13.

[16] Ganusov I, Burtscher M. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proc. the 14th Parallel Architectures and Compilation Techniques*, St. Louis, USA, Sept. 17–21, 2005, pp.350–360.

[17] Zhou H. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proc. the 14th Parallel Architectures and Compilation Techniques*, St. Louis, USA, Sept. 17–21, 2005, Vol.17–21, pp.231–242.

[18] Sun X H, Byna S, Chen Y. Server-based data push architecture for multi-processor environments. *Journal of Computer Science and Technology* (*JCST*), 2007, 22(5): 641–652.

[19] VanderWiel S, Lilja D J. Data prefetch mechanisms. *ACM Computing Surveys*, 2000, 32(2): 174–199.

[20] Oren N. A survey of prefetching techniques. Technical Report CS-2000-10, University of the Witwatersrand, 2000.

[21] Casmira J P, Kaeli D R. Modeling cache pollution. *International Journal of Modeling and Simulation*, May 1998, 19(2): 132–138.

[22] Dundas J, Mudge T. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. International Conference on Supercomputing*, Vienna, Austria, July 7–11, 1997, pp.68–75.

[23] Mutlu O, Stark J, Wilkerson C, Patt Y N. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. the 9th International Symposium on High-Performance Computer Architecture*, San Jose, USA, Feb. 3–7, 2003, p.129.

[24] Doweck J. Inside Intel Core microarchitecture and smart memory access. White paper, Intel Research Website, 2006, http://download.intel.com/technology/architecture/sma.pdf.

[25] Klaiber A C, Levy H M. An architecture for software-controlled data prefetching. In *Proc. the 18th International Symposium on Computer Architecture*, Toronto, Canada, May 27–30, 1991, 19(3): 43–53.

[26] Mowry T, Gupta A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 1991, 12(2): 87–106.

[27] Luk C K. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proc. the 28th International Symposium on Computer Architecture*, Arlington, USA, June 13–15, 2001, 29(2): 40–51.

[28] Liao S, Wang P H, Wang H, Hoflehner G, Lavery D, Shen J P. Post-pass binary adaptation for software-based speculative precomputation. In *Proc. the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Gemany, June 2002, pp.117–128.

[29] Roth A, Moshovos A, Sohi G S. Dependence based prefetching for linked data structures. In *Proc. the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, San Jose, USA, Oct. 4–7, 1998, pp.115–126.

[30] Hassanein W, Fortes J, Eigenmann R. Data forwarding through in-memory precomputation threads. In *Proc. the 18th International Conference on Supercomputing*, Saint Malo, France, June 26–July 1, 2004, pp.207–216.

[31] Collins J D, Wang H, Tullsen D M, Hughes C, Lee Y F, Lavery D, Shen J P. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proc. the 28th Annual International Symposium on Computer Architecture*, Arlington, USA, June 13–15, 2001, 29(2): 14–25.

[32] Collins J, Tullsen D M, Wang H, Shen J P. Dynamic speculative precomputation. In *Proc. the 34th ACM/IEEE International Symposium on Microarchitecture*, Austin, USA, Dec. 2–5, 2001, pp.306–317.

[33] Byna S. Server-based data push architecture for data access performance optimization [Ph.D. Dissertation]. Department of Computer Science, Illinois Institute of Technology, 2006.

[34] Nesbit K J, Smith J E. Prefetching using a global history buffer. *IEEE Micro*, 2005, 25(1): 90–97.

[35] Chen Y, Byna S, Sun X H. Data access history cache and associated data prefetching mechanisms. In *Proc. the ACM/IEEE Supercomputing Conference 2007*, Reno, USA, November 10–16, 2007, Article No. 21.

[36] Chang P Y, Kaeli D R. Branch-directed data cache prefetching. In *Proc. the 4th International Symposium on Computer Architecture Workshop on Scalable Shared-Memory Multiprocessors*, Chicago, USA, April 1994, pp.225–230.

[37] Tran N, Reed D A. Automatic ARIMA time series modeling for adaptive I/O prefetching. *IEEE Transactions on Parallel and Distributed Systems*, April 2004, 15(4): 362–377.

[38] Sun X H, Byna S. Data-access memory servers for multi-processor environments. CS-TR-2005-001, Illinois Institute of Technology, 2005.

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

[40] Jouppi N P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. the 17th International Symposium on Computer Architecture*, Seattle, USA, May 28–31, 1990, pp.364–373.

[41] Jain P, Devadas S, Rudolph L. Controlling cache pollution in prefetching with software-assisted cache replacement. Technical Report TR-CSG-462, Massachusetts Institute of Technology, 2001.

[42] Megiddo N, Modha D. ARC: A self-tuning, low overhead replacement cache. In *Proc. the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, March 31–April 2, 2003, pp.115–130.

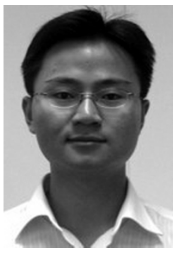
[43] Yang C L, Lebeck A R, Tseng H W, Lee C. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Transactions on Architecture and Code Optimization*, 2004, 1(4): 445–475.

[44] Brown J, Wang H, Chrysos G, Wang P, Shen J. Speculative precomputation on chip multiprocessors. In *Proc. the*

6th Workshop on Multithreaded Execution, Architecture, and Compilation, Istanbul, Turkey, Nov. 19, 2002.

[45] Smith A J. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 1978, 11(12): 7–21.

Surendra Byna received his B. Tech. degree in electronics and communication engineering in 1997 from Jawaharlal Nehru Technological University, India. He received his M.S. and Ph.D. degrees in computer science from Illinois Institute of Technology (IIT), Chicago in 2001 and 2006, respectively. Currently he is a research assistant professor at IIT and a guest researcher at Argonne National Laboratory. Dr. Byna's research interests include high performance computing, data access performance evaluation and optimization, parallel I/O, and multicore data access performance. He is also an owner of www.multicoreinfo.com, which is a portal for multicore related information. More information about Dr. Byna can be found at www.cs.iit.edu/∼suren.



Yong Chen received his B.E. degree in computer engineering in 2000 and M.S. degree in computer science in 2003, both from University of Science and Technology of China. He is currently pursuing his Ph.D. degree in computer science at Illinois Institute of Technology, Chicago. His research focuses on parallel and distributed computing and computer architecture in general, and on optimizing data-access performance, parallel I/O, performance modeling and evaluation in particular.

Xian-He Sun is a professor of computer science and the director of the Scalable Computing Software Laboratory at Illinois Institute of Technology (IIT), and is a guest faculty in the Mathematics and Computer Science Division and Computing Division at the Argonne and Fermi National Laboratory, respectively. Before joining IIT, he worked at DoE Ames National Laboratory, at ICASE, NASA Langley Research Center, and at Louisiana State University, Baton Rouge. Dr. Sun's research interests include parallel and distributed processing, software systems, performance evaluation, and data intensive computing. More information about Dr. Sun can be found at http://www.cs.iit.edu/∼sun/.