Sun XH, Byna S, Chen Y. Server-based data push architecture for multi-processor environments. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 22(5): 641~652 Sept. 2007

Server-Based Data Push Architecture for Multi-Processor Environments

Xian-He Sun[1,2] (孙贤和), Surendra Byna[1], and Yong Chen[1] (陈 勇)

[1] *Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois 60616, U.S.A.*

[2] *Computing Division, Fermi National Accelerator Laboratory, Batavia, IL 60510-0500, U.S.A.*

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

Received March 19, 2007; revised July 4, 2007.

Abstract Data access delay is a major bottleneck in utilizing current high-end computing (HEC) machines. Prefetching, where data is fetched before CPU demands for it, has been considered as an effective solution to masking data access delay. However, current client-initiated prefetching strategies, where a computing processor initiates prefetching instructions, have many limitations. They do not work well for applications with complex, non-contiguous data access patterns. While technology advances continue to increase the gap between computing and data access performance, trading computing power for reducing data access delay has become a natural choice. In this paper, we present a server-based data-push approach and discuss its associated implementation mechanisms. In the server-push architecture, a dedicated server called Data Push Server (DPS) initiates and proactively pushes data closer to the client in time. Issues, such as what data to fetch, when to fetch, and how to push are studied. The SimpleScalar simulator is modified with a dedicated prefetching engine that pushes data for another processor to test DPS based prefetching. Simulation results show that L1 Cache miss rate can be reduced by up to 97% (71% on average) over a superscalar processor for SPEC CPU2000 benchmarks that have high cache miss rates.

Keywords performance measurement, evaluation, modeling, simulation of multiple-processor system, cache memory

1 Introduction

The emergence of chip multiprocessing (CMP) architectures has increased the peak CPU performance significantly. High end computing (HEC) machines with Petaflops of computing power is in the near horizon. However, disparity among processors, storage, memory, network, and applications causes a gap between peak performance and sustained system performance, and this gap is growing rapidly[1] (see Fig.1). While CMP technology is becoming the driving technology in increasing computing power, it does not provide any direct solution to reduce the gap rather than accelerating the enlargement of the gap. HEC machines' sustained performance is a low single digit percentage of their peak performance is common. Multi-core concurrent processing often achieves lower performance than expected, due to poor utilization of additional processing cores. With technology advances, especially with the emergence of CMP technology, computing power is no longer the bottleneck of sustained system performance, but the data access performance is. In this study, we introduce a server-push data architecture, which trades computing power with data access delay to reduce the performance disparity between processors and data access, ultimately to increase the sustained system performance.

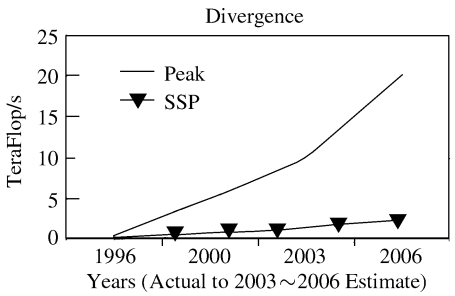

Fig.1. "Divergence problem"—Increasing gap between peak performance and sustained system performance (SSP) (Source: [1]).

The advance of computing and memory technologies is unbalanced. Following the Moore's law, CPU performance has been improving rapidly (52% until 2004 and 25% since then[2]), while memory performance only has been improved 7% per year on average during the last 20 years. This leads to the so called "memory wall"[3]. Caching and prefetching are the commonly used methods to mask the performance disparity between computing and data access

Regular Paper

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

performance. Caching holds data temporarily, while prefetching fetches the data to a cache closer to the computing processor before it is requested. Various prefetching strategies have been proposed and developed during the years[4~8]. However, their performance varies largely from application to application, and is generally poor on HEC computers. The poor performance of current prefetching technology may be due to different reasons. One noticeable reason is that current prefetching is based on client-initiated prefetching, where the computing processor initiates prefetching. While letting a computing processor prefetch required data sounds to be a straightforward solution, client-initiated prefetching has many limitations. For instance, predicting what data to fetch require computing power, aggressive (accurate) prediction algorithms require more computing power, which leads either to untimely prefetching (poor prefetching accuracy) or to degrade computing process performance. In some cases, predicted prefetching instructions are given lower priority than original load/store instructions. This again leads either to untimely prefetching or to wastage of computing power spent on predicting future accesses. In chip-level multiprocessing, multiple cores share cache memories and memory bandwidth, which puts even more pressure on data access if multicore concurrent processing is conducted. These limitations have to be overcome to improve the effectiveness of aggressive prefetching algorithms.

Recognizing the limitation of client-initiated prefetching and taking advantage of the abundant computing power of chip multiprocessors have led to several recent proposals of new prefetching strategies on multi-core processors[9~16]. The main idea of these approaches is to let a helper thread run ahead of the program main thread on a separate core to initiate load cache misses in a multicore machine. Pre-execution can be run on a core close to the program's main thread, which is called *pull-based* pre-execution, or on a memory processor, which is called *push-based* prefetching[11,17]. The former fits the current multicore chip architecture well. The latter requires special hardware support but is more efficient in masking data access delay and implements decoupling of data access from computing[18].

In this study we extend the concept of decoupling of computing and data access even further. We propose a design for data access server, named Data Push Server (DPS), which is dedicated to predict data access pattern and push data closer to computing processors in time. Here the term "push" means that, unlike traditional client-initiated prefetching, DPS issues prefetching instructions on behalf of computing cores. DPS does not execute any computing related instructions. Its whole purpose is to provide data push service and to prefetch based on data access prediction. This further separation of computing and data service has several benefits. First, a dedicated server can adapt to complex prediction algorithms for more aggressive prediction and can push data into multiple computation threads or cores. This is especially beneficial for HEC, where parallel processing is often achieved with the Single Program, Multiple Data (SPMD) model[19]. Second, DPS is flexible to choose strategies dynamically to predict future accesses based on data access history. Instead of looking for a single algorithm to predict all data access patterns, which does not exist, DPS can adaptively choose a prediction method based on the history of accesses and compiler hints. This, again, is very beneficial to HEC where few of the so called "grand challenge applications", which often run repeatedly. Third, DPS uses temporal data access information to predict *when* to push data. This gives an opportunity to modify prefetch distance adaptively and to avoid costly synchronization needed for pre-execution strategies to initiate prefetching *in time*. Lastly, using a dedicated server for push-based prefetching gives an opportunity for using hints from previous execution of an application. All these benefits make push-based prefetching a viable method for masking data access cost more effectively.

DPS can be implemented at various levels of a memory hierarchy[20]. DPS for multicore computer is designed carefully and presented in this study. We address technical issues, such as what data to prefetch, when to fetch, and how to push. A new data access prediction algorithm is also introduced. The SimpleScalar simulator[21] is modified to include DPS prefetching engine. While the simulation results are preliminary, results on benchmarks and compact applications show that DPS has merits and has a real potential.

The organization of this paper is as follows. In Section 2, we present the structure of DPS and discuss the functionality of various components of DPS. In Section 3, we present our initial performance results. In Section 4, we discuss related work, followed by conclusions and future work in Section 5.

2 Data Push Server

Fig.2 shows the structure of Data Push Server (DPS). The goals of DPS are to predict data access patterns of applications and to push the predicted data from main memory to a cache closer to processor. Its three primary components are: pattern detection manager, prefetch engine, and management engine. The *Pattern Detection Manager* (PDM) collects history of data accesses in spatial and temporal dimensions. Data access information in spatial dimension includes

the strides between successive accesses. Information in temporal dimension refers to the time of accesses, either in clock cycles or inter-reference distance. The PDM then classifies patterns of those data accesses[22]. The prefetch engine is responsible to predict future accesses and the timing. It in turn has three subcomponents: prefetch strategy selector, prefetch predictor, and request generator. The *Prefetch Strategy Selector* (PSS) adaptively selects an appropriate method to predict future accesses based on the pattern information. The *prefetch predictor* of the prefetch engine decides *what* data to fetch and the *request generator* decides *when* to push data so that the prefetched data arrives at its destination *in time*. Here by "*in time*", we mean that data is pushed from its source to destination within a window of time before it is required, and where it does not replace other data blocks from cache falsely. By moving data into a cache too early, it may replace data blocks that would be accessed in the near future. Our strategy aims to avoid such negative effects. The management engine is responsible to issue instructions to push data. The prefetch requests are kept in a prefetch queue and *data propeller* in the management engine issues a signal to push the data to its destination. The source of data in multicore processor environment is main memory, and the destination is cache memory. In the following subsections, we discuss the functionality of DPS components in detail.

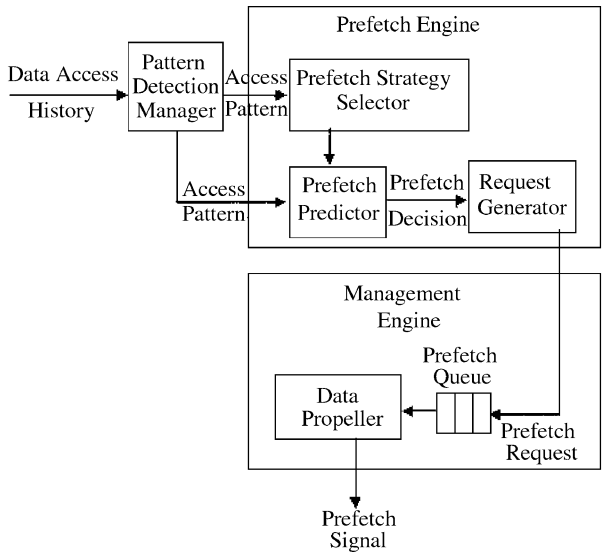

Fig.2. Components of Data Push Server.

2.1 Data Access Pattern Detection

In research literature, there are many strategies to predict future data references. However, no single strategy accurately predicts all data access patterns. Sequential and strided strategies can predict regular constant and varying strided accesses, while another set of strategies try to chase pointers and data structure traversals[23~25] that require compiler and user provided hints. Pre-execution based approaches[11,26] often use a helper thread to run slices of code to predict future accesses. Complexity of these strategies varies. Using simple strategies cannot capture complex patterns and complex strategies suffer from high overhead in predicting simple access patterns. An accurate prefetching mechanism should support various prediction strategies and should adapt to data access patterns of an application at runtime.

In DPS, the *Pattern Detection Manager* (PDM) detects data access patterns, and the *prefetch strategy selector* selects an appropriate prediction strategy based on the detected pattern. To detect whether a pattern is formed by simple strides or complex variable strides, the PDM observes the distances (spatial and temporal strides) between consecutive data references. We classify data access references into contiguous, non-contiguous, and combinations of contiguous and non-contiguous patterns[22]. We divide these patterns further based on repetition of occurrence of each pattern and on variation of strides between non-contiguous patterns. Based on this classification, the PDM characterizes a pattern and passes that information to the *prefetch strategy selector*.

2.2 Predicting Future Data References

The *prefetch strategy selector* (PSS) chooses a prediction strategy based on initial information regarding a pattern. Data access patterns can be regular or random. Regular patterns can be simple strided or complex combinations of nested strides. Complex regular strided patterns usually exist in accessing arrays of structures, where variables of structures are of multiple basic data types. Some data access patterns are accessed only once, while other access patterns appear repeatedly. Such repeating patterns are common when loops or functions execute repeatedly. Many strategies exist to predict future references with constant strides or patterns of strides[4,6,8]. However, patterns with variable strides and repetitions need more analysis to find regularity among them. With dedicated machine, as computing power is available for prediction, we introduce a new method that predicts regular patterns with constant stride as well as variable stride accesses and repeating patterns. This prediction strategy is based on a finding "*what number comes next*" in the context of number sequences[27]. This method forms a *difference table* of depth d, which we call *multi-level difference table* (*MLDT*). Existing strided prefetching[4,6] and distance prefetching[8] methods use the distances (strides) between successive

block numbers up to one level to find regularity. In MLDT scheme, we extend finding distances for more than one level. Each entry of the *difference table* is the difference between the two entries just above it (in the sense "right entry minus left entry").

Fig.3 shows a pattern of successive data references. The first differences ($d = 1$) are the strides between the right reference minus the left reference. If these strides are different, differences among these strides are calculated. Second differences ($d = 2$) in Fig.3 are equal to a constant value of 2. After a constant difference is found, the next entry of second differences above can be predicted to be the same. As shown in Fig.3, third entry of second differences is predicted as 2 (predicted entries are marked in bold face font). This is added to the third entry of the first differences, i.e., $7 + 2 = 9$. This value is added to the fourth entry of references to find the fifth reference, i.e., $16 + 9 = 25$. The future references are predicted (36, 49) in the above example.

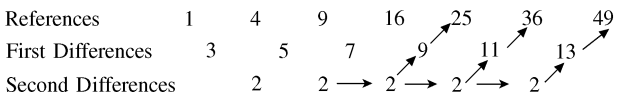

Fig.3. Multi-level Difference Table for variable stride non-contiguous pattern.

In the example above, we have shown finding the address of next data block. We have worked out to find polynomials to predict the next k-th reference in a reference sequence. In Fig.4, we show a three-level difference table. The references are represented by A_i ($i = 0$ to n). The first differences are B_i ($i = 0$ to $n - 1$), second differences are C_i ($i = 0$ to $n - 2$), and third differences are D_i ($i = 0$ to $n - 3$). We present these polynomials up to the depth of three, which can be extended further.

n	0	1	2	3	4	5	6
References	A_0	A_1	A_2	A_3	A_4	A_5	A_6
First Differences		B_0	B_1	B_2	B_3	B_4	B_5
Second Differences			C_0	C_1	C_2	C_3	C_4
Third Differences				D_0	D_1	D_2	D_3

Fig.4. Example of multi-level difference table (MLDT).

If the depth of a difference table is 1 and if a constant value can be found among the first differences, i.e., in Fig.4, B_i (where $i = 0$ to $n - 1$) is the constant B, value of the k-th reference from reference A_r can be found with the following formula.

$$A_{r+k} = A_r + k \times B.$$

For a difference table of depth 2, where second differences are constant, value of the k-th reference from reference A_r can be found with the following formula.

$$A_{r+k} = A_r + k \times B_{r-1} + \frac{k \times (k+1)}{2} \times C.$$

For a difference table of depth 3, where third differences are constant, value of the k-th reference from reference A_r is:

$$A_{r+k} = A_r + k \times B_{r-1} + \frac{k \times (k+1)}{2} \times C_{r-2} + M_k D.$$

Here $M_k = k/6 \times (k-1) \times (k-2) + k^2$, where $k = 1, 2 \ldots$.

MLDT works similar to existing stride based prefetching strategies that predict constant stride patterns. In addition, this method finds sets of repeating differences, and ultimately finds the actual pattern in accessing structures with variable strides data access pattern. When the depth of a difference table is 1, references have a constant stride. Existing stride-based prefetching strategies are effective for this special case. Strides similar to Fig.3 can be seen in image rendering applications. For variable strided patterns, MLDT searches for regularity among data references by finding deeper difference table. We extend this method to find repeating sets of strides (e.g., 4, 8, 4, 4, 8, 4, 4, 8, 4, ...) at each level of difference table. These repeating sets of patterns are common in accessing user-defined datatypes (e.g., structures). For patterns with various strides among multiple variables, the prefetch predictor requires more time to learn data access patterns. This can be extended to time series analysis such as ARIMA models[28] to make complex pattern predictions.

Identifying data access pattern of irregular accesses is complex, which may be impossible to predict the addresses of future accesses. The overhead during pattern learning phase of such data accesses can be reduced by utilizing compiler hints and application support[24,25]. Most of current compilers perform highly sophisticated data dependence analysis. From this analysis, compilers can generate hints to derive data access patterns. Similarly, the application developers can provide hints regarding the data access pattern in their applications. Runtime profiling of the application also provides such hints. In this paper, we limit the scope to find regular, complex regular and repeating patterns. Dynamic selection of multiple strategies based on compiler and user-provided hints is left as future research work.

2.3 Predicting the Time to Prefetch

The issue of *when* to prefetch in the existing methods is limited by the occurrence of an event such as a cache miss or a page fault (prefetch on miss) or the first access to a data block (tagged prefetch) etc. However, these strategies do not guarantee that the prefetched data will reach its intended destination "*in time*" to overlap the processor stall time. The efficiency of prefetching in time depends on three factors

(Fig.5): the time to predict future accesses (T_{pred}), the latency of initiating and transferring data from its source to destination (T_{lat}), and the gap between current time and the next data reference that would cause a demand cache miss (T_Δ) when no prefetching is applied. If T_{miss} denotes the penalty caused by a cache miss, prefetching can take place in the following situations.

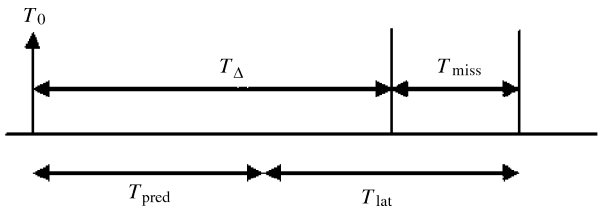

Fig.5. In time prefetching.

- *Case* 1. If $(T_{\text{pred}} + T_{\text{lat}}) > (T_\Delta + T_{\text{miss}})$, the prefetching is completely useless.
- *Case* 2. If $(T_{\text{pred}} + T_{\text{lat}}) > T_\Delta$ and $(T_{\text{pred}} + T_{\text{lat}}) < (T_\Delta + T_{\text{miss}})$, there is a partial gain of performance improvement based on how much of T_{miss} is overlapped.
- *Case* 3. If $(T_{\text{pred}} + T_{\text{lat}}) = T_\Delta$, the prefetching is *in time* and the prefetching is the most effective (Fig.5).
- *Case* 4. If $(T_{\text{pred}} + T_{\text{lat}}) < T_\Delta$, there are two cases.
  - A. If the destination of prefetched data has empty space to accommodate, the prefetching has no negative effect.
  - B. If the destination of prefetched data is full, a victim cache line has to be replaced based on replacement policies. This negative effect of prefetching may result in extra cache misses.

To benefit from prefetching, a prefetching strategy has to be adaptive to decide if a prefetch would be useful or not. A useless prefetch increases traffic of the bus, and may pollute a location on the destination of that prefetch. This necessitates the prediction of T_Δ to make a decision whether to prefetch or not.

In DPS (Fig.2), the *request generator* decides when to prefetch. The request generator varies the value of k (from Subsection 2.2) based on the detected spatial and temporal data access history of a cache. Temporal history contains clock ticks of processing core to recognize its timing pattern. The request generator predicts T_Δ and adjusts the value of k so that $(T_{\text{pred}} + T_{\text{lat}})$ is approximately equal to T_Δ. We assume that only one application runs on a processing core at a time, since it is complex to observe temporal pattern of data accesses when multiple tasks are running on the same core. We currently use MLDT method to identify temporal pattern in order to predict T_Δ. In the future we plan to use ARIMA models to predict temporal access patterns.

2.4 Pushing Data

The *data propellor* component of DPS delivers data to processing units. After predicting the addresses of future references by the prefetch engine, the data at these addresses has to be delivered to appropriate processing units. In traditional hardware prefetching strategies, prefetching instructions are issued by the same processing unit that executes a program. In DPS strategy, the predicted future data references are stored in a prefetch queue. The prefetch engine sends this prefetch queue to the data propellor, and the data propellor issues prefetching (push) instructions to move the data from the memory to processing units that need data. Special hardware support is needed to issue instructions to push data.

2.5 Suggestions for Architectural Support

In order to implement DPS and to obtain the benefits of aggressive prediction strategies, special hardware is needed to support the implementation of DPS on multi-core processors. DPS requires to collect data access information from processor cores in order to recognize their data access pattern. For instance, in a multicore processor, DPS core collects data access history of the processing cores. A likely implementation is shown in Fig.6, where a Data Access History buffer is located on each core. This buffer is similar to Prefetch History Table of Intel Core microarchitecture[29], but with more fields to support in time prefetching. DPS requires hardware support to access this DAH buffer to analyze the history to predict future accesses. DPS also requires hardware support to push data from memory to upper level cache of the processing cores.

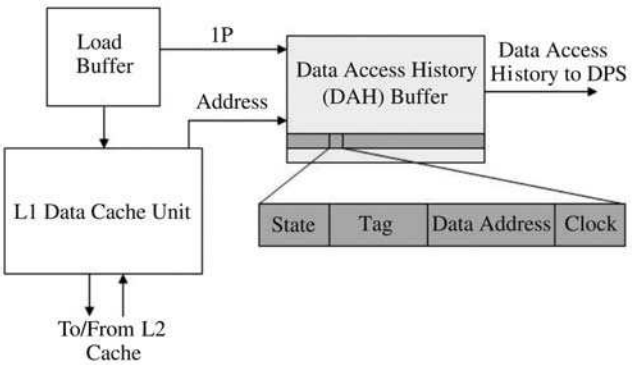

Fig.6. Data Access History collection for DPS.

DPS sends prefetch signal to main memory to push data into L1 level cache of the processing cores. Existing multicore processor architectures do not have such

support to perform these two operations directly. The current cores of processors can issue prefetch instructions to fetch data closer to their own core, which supports client-initiated prefetching. To implement DPS, support to push data for other cores is required.

When data is moved closer to a processing core, prefetched data is validated if it is already in the cache memory. Duplicate copies need to be discarded, which is necessary to maintain coherence if data has already been modified. We propose to use a separate prefetch cache, which can be accessed concurrently with L1 cache. A separate prefetch cache to store prefetched cache lines is not new. Among current processors, UltraSPARC-IIIi and IV provide a 2KB data prefetch cache[30]. Such provision reduces replacement of data from L1 or L2 caches. From CPU's memory requestion, after physical address mapping from TLB is obtained, L1 cache and prefetch cache tags are compared simultaneously. As the processor core searches data cache and prefetch cache in parallel, the server-based data push model benefits more by reducing data cache misses further. With the potential of DPS, we suggest that provision of hardware support benefits the overall performance.

Emerging chip-level multiprocessors show prospect to implement DPS. There are numerous announcements for developing processors with more than 1000 cores. The cores of an IBM Cell processor[31] have an internal bus, which can be used for observing patterns of their local memories and for pushing data directly to their local memory. Newly proposed multicore and manycore architectures support heterogeneous core design, where computing cores have different functionality. DPS can be implemented as one of such specialized core, that performs prefetching for other computing cores. The rapid growth of field-programable gate arrays (FPGA) and reconfigurable computing is also promising for developing specialized cores such as DPS.

3 Experimental Results

3.1 Simulation Methodology

We evaluate performance of DPS using an extended version of the SimpleScalar toolset V4.0[21]. The baseline simulator configuration consists of a four-issue dynamic superscalar cores similar to that of Alpha 21264, with configuration shown in Table 1. To apply strided prefetching, we modified the *sim-outorder* simulator using a 512 entry reference prediction table (RPT) and prefetch instructions are triggered when a cache miss occurs. The prefetch distance is constant and set as 8 for strided prefetching. Although prefetch distance of 8 is not optimal for all applications, Puzak *et al.*[32] suggest that prefetch distance above 5 branches ahead shows better performance and prefetch distance beyond 10 branches has little benefit. To analyze data access history collected in Data Access History buffer on processing core (Fig.6), we use a Data Access History (DAH) table structure. The DAH table has a tag, count, tail and head pointer fields (Fig.7). Tag field records the instruction address. Each entry is a doubly linked list, which is a queue and keeps track of data access addresses and the time of occurrence (in cycles) of the corresponding entry instruction. Fig.7 illustrates a constant stride is detected for instruction

Table 1. 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
DAH Size	512 entries
Prefetch Queue	512 entries

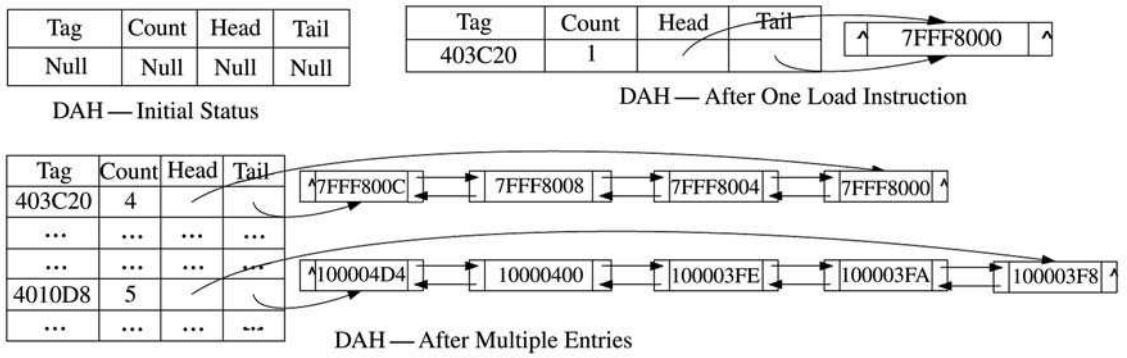

Fig.7. Data Access History (DAH) table.

address 403C20 and a variable stride (a pattern with depth 2) is detected for instruction address 4010D8. This design makes DAH capable of capturing more history of recent accesses instead of only two latest accesses as in RPT, thus makes it possible to capture multi-level difference table of length n.

To simulate DPS core, we modified the *sim-outorder* simulator to add another core. All the components of DPS prefetching engine run on this core. Operation of this core does not affect the cycles or instructions of the processing core. We added the prefetching to the main components of SimpleScalar simulator to observe data access patterns of processing core and to predict future references. A pattern detection manager (PDM), a DAH table, and a prefetch queue, similar to the ready queue structure of *sim-outorder* simulator, are implemented. The PDM collects data into DAH table. The prefetch strategy selector chooses either simple strided prefetching strategy or the MLDT strategy based on the pattern information provided by the PDM. The DPS core triggers a prefetch when there are prefetch requests in the prefetch queue. We modified the memory module of the DPS core to introduce an instruction to prefetch data into the L1 cache of processing core. We modified the cache manager of processing core to support data push by the DPS core. We also have modified the simulator to include a bus to support collecting DAH buffer information and pushing data into L1 cache of the processing core. This bus dedicates part of its bandwidth for prefetching while the rest for normal operations.

3.2 Experimental Setup

As a first assessment of the potential of server-based data push model, we constructed a set of simple and complex regular strided pattern benchmarks. For these benchmarks, we analyzed the cache performance. We then verified the performance improvement of SPEC CPU2000[33] benchmarks that have poor L1 cache performance. Table 2 lists microbenchmark kernels that are crucial components of well-known benchmarks such as BLAS (Basic Linear Algebra Subroutines)[34], STREAM[35], and SPEC CPU2000 benchmarks. 2D-matrix transpose and 2D-matrix multiplication are important matrix operations in scientific applications. These two operations exist in many benchmarks that test the performance of computer architectures including BLAS, CPU2000 benchmarks. *Struct* kernel is taken from CPU2000 benchmarks, which has complex regular pattern. Fig.8 shows another nested strided pattern, which accesses a 3-dimensional matrix. This pattern contains repetition of three different strides. The fist stride contain accessing two contiguous elements of 1-dimensional array. The second and third accesses contain accessing two 1-dimensional and 2-dimensional arrays, respectively, with different strides.

Table 2. Benchmark Kernels

Kernel	Operation	Access Pattern

2D-Matrix Transpose	**for** $(i = 0; i < N; i++)$
	  for $(j = 0; j < N, j++)$
	  $y[i][j] = x[j][i];$
		y: Contiguous
		x: Non-Contiguous

2D-Matrix Multiplication	**for** $(i = 0; i < N; i++)\{$
	  for $(j = 0; j < N; j++)\{$
	  $t = 0;$
	  for $(k = 0; k < N; k++)\{$
	  $t+ = a[i][k] * b[k][j];$
	  $\}$
	  $c[i][j] = t; \} \}$
		a: Contiguous
		b: Non-Contiguous
		c: Contiguous

Struct Accesses	**for** $(i = 0; i < N; i++)\{$
	  type_a$[i]->$ longval1$=a[i];$
	  type_a$[i]->$ longval4 $= b[i];$
	  type_a$[i]->$ longval8 $= c[i];$
	  $\}$
		type_a: Non-Contiguous, Irregular Stride of Repeating 1, 64 and 64, a, b, c: Contiguous

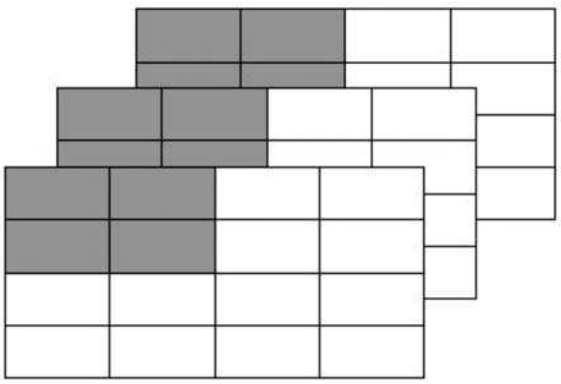

Fig.8. 3-dimensional nested strided data access Innermost stride to access 1-D array, second strided pattern to access 2-D array, and the outermost strided pattern to access 3-D array.

We select these benchmarks, as they represent data access patterns found in real codes and to explain how DPS prefetching works. For instance, Matrix transpose and multiplication operations are classic examples of noncontiguous (strided) accesses that contribute to high cache miss rates when the data size exceeds the cache size. These algorithms have been the targets of numerous cache performance improvement studies. A *struct* is a user-defined datatype in C language. These *struct* accesses represent variable strided data accesses, when they are defined with different basic data types or with other user defined data structures. These accesses increase the cache misses when the stride between successive accesses is larger than a cache line.

We compare the L1 cache miss rates of all bench-

marks for three cases: without prefetching (base case), with strided prefetching, and with DPS prefetching strategy. In the base case (without prefetching), the cache misses include compulsory, capacity and conflict misses. The strided prefetching strategy predicts the next stride based on the history of recent accesses and a prefetch instruction is issued only on the occurrence of a cache miss. These programs were compiled with gcc V3.2.3 with optimization flags turned off to exclude the effect of compiler optimizations. We use the SimPoint[36] toolset to select a representative starting point beyond a program's initialization phase.

We also evaluate the performance of SPEC CPU2000 benchmarks. We selected five benchmarks that have high L1 cache miss rate. These programs were compiled with `gcc V3.2.3` using "`-O3-static`". Each program is simulated for 200 million instructions after fast forwarding past the initialization phase selected by the SimPoint toolset.

3.3 Performance Comparison

Fig.9 shows L1 cache miss rates of the benchmark kernels. We set $N = 1024$ in all the benchmark kernels, where each loop iterates for N times. 2-D matrix transpose has one contiguous access pattern (array y), and one non-contiguous access pattern (array x). When data size is bigger than cache size, each cache line is loaded into cache while accessing a matrix column is flushed (with row-major order) before it is reused in accessing the next column. The cache miss rate without prefetching is 56.25%. Using strided prefetching, the cache miss rate is reduced to 26%[①]. In accessing array x, there are two types of strides: forward (positive) strides to access each element of a column of the array, and a backward (negative) stride after transposing a column fully. In strided prefetching strategy, prefetches are initiated only on a cache miss. With DPS strategy, the request generator adjusts timing to prefetch (i.e., prefetch distance is selected dynamically), where miss rate is reduced to 0.01%. These misses occur during the data access pattern learning stage. L1 cache miss rate of 2-D matrix multiplication without prefetching is high due to the number of non-contiguous accesses to array b (∼50%). Here, arrays a and c are accessed contiguously, but strides are different. Array a is accessed in every iteration, while array c is accessed once every N iterations. Each of these arrays has reuse among the fetched cache lines, but array b has no cache reuse, when data size is bigger than cache size. The strided prefetching strategy reduces the miss rate to 24%. For struct accesses benchmark, the strides are set to 1, 64 and 64 and these strides repeat. The L1 cache miss rate with base case is 55%. In this case DPS prefetch strategy is able to predict repeated sets of patterns. The request generator adjusts the value of k, as there is no reuse in accessing *type_a* structure. With this strategy, most of the cache misses are overlapped. Similarly, for 3-D access pattern, DPS prefetching is able to predict repeated sets of patterns, without requiring multiple learning stages for each outer loop iteration.

Fig.10 shows L1 cache miss rates of CPU2000 benchmarks. With DPS prefetching, L1 miss rates are reduced significantly for all the benchmarks. These five benchmarks have a large number of nested loops that access user defined data types (*struct*), where data access exhibit complex regular patterns. For *ammp* L1 miss rate reduction is 97.05%. For *applu* it is 48.9%, for *art* it is 96%, for *mcf* it is 32%, and for *mgrid* benchmark it is 66.5%. These miss rates are 40% to 95% less (66% on average) compared to strided prefetching. Fig.11 compares the number of L2 misses for these benchmarks, which shows significantly reduced number of misses with DPS prefetching.

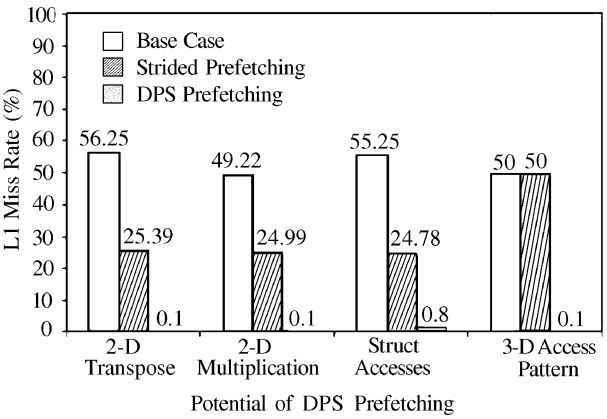

Fig.9. Performance of Kernel benchmarks.

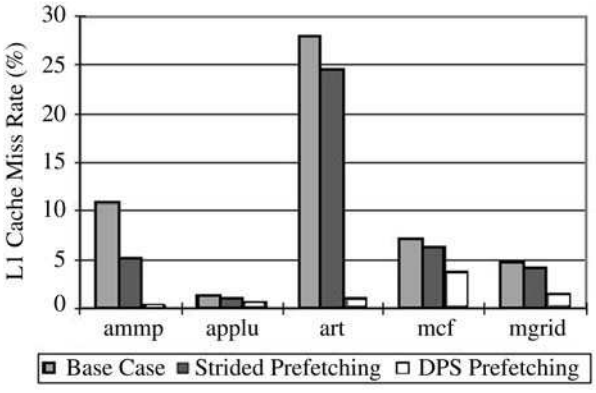

Fig.10. L1 miss rate for SPEC 2000 benchmarks.

[①]In this experiment, compiler optimizations are turned off to verify the improvement of strided prefetching alone.

The cost of aggressive prefetching algorithms could be high. Fig.12 shows the values of IPC (instructions per cycle) improvement for the above CPU2000 benchmarks. The first bar shows the IPC improvement with strided prefetching. The second bar represents the IPC improvement when we implement DPS prefetching without a dedicated DPS core, i.e., DPS prefetching is implemented on the same processing core, where benchmark code is running. The third bar represents the IPC improvement, when we use a dedicated DPS core for our prefetching strategy. Strided prefetching improves IPC slightly, but degrades for *applu* benchmark. When DPS is implemented on the same processing core, the IPC improvement is negative for all benchmarks except for *ammp* benchmark. This shows that, even though aggressive DPS prefetching is effective in predicting future references based on the history of accesses, when it is implemented on the same processing core, the overall performance degrades. With the use of a dedicated memory server core, the IPC values improve significantly, benefiting from aggressive prefetching.

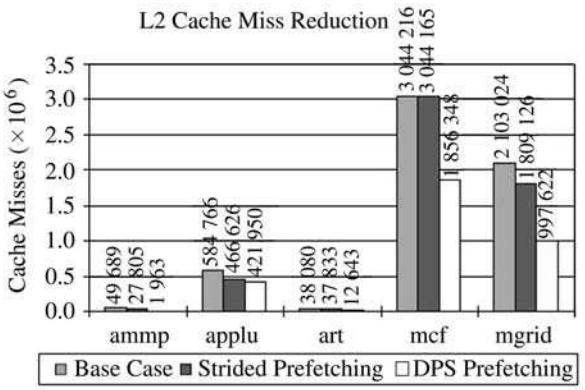

Fig.11. L2 misses for SPEC 2000 benchmarks.

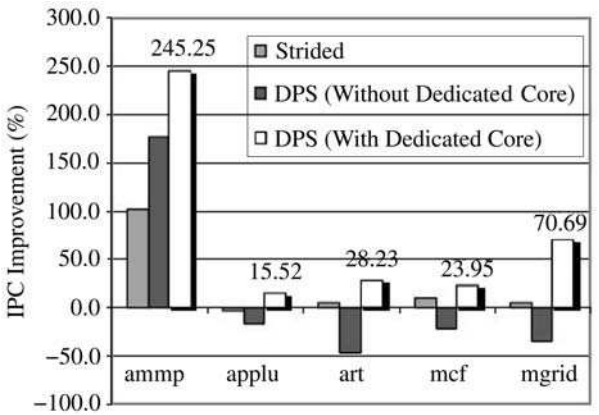

Fig.12. IPC improvement with DPS prefetching.

Cache pollution is a negative effect of aggressive prefetching. The cache replacement rate reflects cache pollution and it increases if data is not pushed in time. Fig.13 shows the replacement rates for the above CPU2000 benchmarks. It can be observed that the cache pollution effect is none for all the benchmarks with our aggressive DPS prefetching strategy. For strided prefetching, the data is prefetched only when there are regular strided patterns among data accesses, which does not increase cache pollution and cache performance improvement is also low. With aggressive DPS prefetching, cache performance improvement is higher, while keeping cache pollution low.

These performance results illustrate the potential of using a dedicated DPS core for prefetching. In actual implementation, the observation of data access patterns at processing cores may involve some overhead. The use of a DPS core reduces the actual prefetching overhead at processing cores and the performance gain would supercede the overhead involved in observing the patterns. We plan to study these costs in the future. Moreover, DPS has flexibility to choose prediction strategies adaptively, to prefetch data *in time* and to serve multiple clients. These functionalities of DPS broaden the impact of CMP architectures and in bridging the divergence gap of High-end Computing.

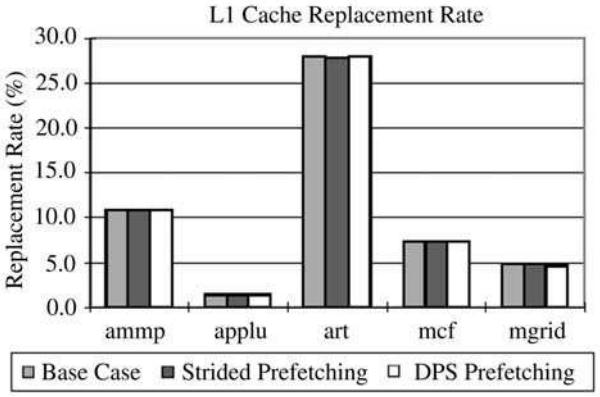

Fig.13. Effect on replacement rate with DPS prefetching.

4 Related Work

Data prefetching is a well studied research area of computer architecture. Traditional hardware data prefetching strategies on single core processors range from simple sequential prefetch strategies to complex Markov prefetching, and to using compiler hints in prefetching and chasing pointers. Sequential strategies[5,37] prefetch next k lines of data, while strided strategies[4,6~8,38] predict future strides based on past accesses. These strided strategies propose to issue prefetching instructions during some event (i.e., on cache miss, on each memory access, on accessing

prefetched data, on branch). However, with the increasing complexity of these methods, the benefits of prefetching diminish in the traditional client-initiated prefetching. Software-controlled prefetching[39] gives control to a developer or a compiler to insert prefetching instructions into programs. Many processors provide support for such prefetching instructions in their instruction set. However, software-controlled prefetching puts burden on developers and compilers, and is less effective in reducing memory stall time on ILP processors due to late prefetches and resource contention[40].

With the emergence of multithread support in processors, many thread-based solutions have been proposed to deal with the complexity issue. These methods can be roughly classified into two categories: pre-execution based and prediction based. Pre-execution based methods often use a helper thread to run slices of code ahead of main thread. A small list of various proposals using pre-execution include Luk *et al.*'s software controlled pre-execution[14], Liao *et al.*'s software-based speculative precomputation[13], Zilles *et al.*'s speculative slices[16], Roth *et al.*'s data-driven multithreading[15], Annavaram *et al.*'s data graph precomputation[23], and Hassanein *et al.*'s data forwarding[11]. Many of these methods often rely on compiler support to select slices of code to pre-execute and to trigger execution of that code. Collins *et al.*[10] suggest using hardware to select instructions for pre-computation. Zhou[41] and Ganusov *et al.*[26] proposed utilizing idle cores of a CMP to speed up single threaded programs. Zhou's *dual-core execution* (DCE) approach uses idle core to construct large, distributed instruction window and Ganusov *et al.*'s *future execution* (FE) uses idle core to pre-execute future loop iterations using value prediction. In contrast to pre-execution approaches, DPS resides on a dedicated data server and adaptively chooses future data prediction strategies aggressively. DPS is designed to serve multiple processing cores simultaneously, where as DCE and FE are tightly coupled to one core. DPS predicts temporal pattern to provide in-time prefetching, while pre-execution approaches require synchronization to achieve that.

Prediction based multi-threaded strategies use helper threads to predict future references based on history of past accesses. Solihin *et al.*[42] propose memory-side prefetching (similar to *push-based* prefetching), where a memory processor is designed to reside within main memory. This memory processor observes history L2 cache misses and predicts future accesses. This scheme uses stride-based and pair-based correlations among past L2 cache misses and pushes predicted data to L2 cache. Our DPS strategy suggests using a dedicated server outside the main memory to observe data accesses at L1 cache level and to push predicted data to L1 and L2 caches. Based on these data accesses, DPS has flexibility to choose prediction strategy and to serve multiple processing cores. DPS also predicts *when* to push data based on temporal pattern of data accesses for in-time prefetching. Hassanein *et al.*[11] also use memory-side prefetching and suggest to forward data to either L1 cache or directly into CPU registers, but their scheme is based on pre-execution.

Among the above related works, there are many prediction algorithms that are more aggressive than simple strided prefetcher, which can improve prediction of future accesses based on history of accesses. However, as we have shown in Fig.12, the complexity of aggressive prefetching algorithms has a negative effect on overall performance. This has been a major reason that many processor architectures still stick to a simple strided prefetcher and do not develop aggressive prefetching algorithms that are proposed over the last two decades. Our proposed server based method is a possible solution by providing some dedicated hardware for prefetching. This dedicated prefetch engine can use any type of prediction algorithm or a combination of prediction algorithms. Through push-based prefetching aggressive prediction algorithms have a chance to be used if their complexity of moved to another core.

The benefits of DPS are extendable to multiprocessor environments such as SMP, where nodes share the same memory. DPS fits well as a memory server in these environments. DPS pushes data from the shared memory to local memory of the compute nodes. Since the server-based push model separates data movement from computing, its impact is fundamental and is beyond the field of HEC. For instance, it can serve as the μ *proxy* between the file server and its clients in a distributed file system to improve scalability; can enhance coherence to provide a single image in a parallel system; and can virtualize storage in a Grid environment. Even in HPC, DPS can be enhanced in language, compiler and scheduling, and can be implemented at system or application level.

5 Conclusions and Future Work

In this study, we have presented the server-based data push architecture, called Data Push Server (DPS), for masking processor stall time effectively. DPS uses a data server in parallel with processing core (or cores) to predict future data accesses and to push the required data to its destination *in time*. A structured design is presented to implement DPS in multiprocessor machines. A novel aggressive prefetching algorithm is also proposed, to predict constant strided,

varying strided and repeating patterns.

Initial simulation results show that DPS has a profound potential to improve the memory access performance of various data access patterns. DPS has reduced L1 cache miss rates of benchmark kernels with various strided patterns, in particular those of SPEC CPU2000 benchmarks to less than 1%. This is a significant improvement (up to 95%) over strided prefetching. These results show the potential of DPS in avoiding most of the processor stall time by moving data closer to computing in time.

We have only demonstrated potential performance gains of DPS in this study. Many research issues require more investigation. We plan to study DPS approach further for fast data access and to explore its potential in other domains of information processing. We plan to extend this work to study detailed implementations of DPS and to design a strategy to select various pattern prediction strategies based on compiler and user-provided hints. This will improve the effectiveness of DPS in predicting irregular patterns such as data structure traversals. We would like to continue studying the usage and scalability of DPS core to support multiple processing cores as well. We intend to explore more accurate pattern prediction algorithms, such as time series analysis models. Emerging heterogeneous multicore architectures and advances in reconfigurable computing show promise to implement our DPS as a specialized core that prefetches data for other computing cores.

References

[1] DARPA. High productivity computing systems (HPCS), vision: Focus on the lost dimension of HPC — "User & system efficiency and productivity". http://www.darpa.mil/ipto/programs/hpcs/vision.htm.

[2] John Hennessy, David Patterson. Computer Architecture: A Quantitative Approach. Fourth edition, Morgan Kaufmann, ISBN: 0123704901, 2006.

[3] Wm A Wulf, Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARPH Computer Architecture News,* March 1995, 23(1): 20∼24.

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

[5] Dahlgren F, Dubois M, Stenström P. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proc. International Conference on Parallel Processing* (*ICPP*), Los Alamitos, CA, USA, CRC Press, 1993, Vol.1, pp.56∼63.

[6] Fu J, Patel J H. Data prefetching in multiprocessor vector cache memories. In *Proc. the 17th Annual International Symposium on Computer Architecture*, Toronto, Canada, 1991, pp.54∼63.

[7] Joseph D, Grunwald D. Prefetching using Markov predictors. In *Proc. the 24th International Symposium on Computer Architecture,* Denver-Colorado, 1997, pp.252∼263.

[8] Gokul Kandiraju, Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *Proc. the International Symposium on Computer Architecture*, Anchorage, Alaska, 2002, p.195.

[9] Alexander T, Kedem G. Distributed predictive cache design for high performance memory system. In *Proc. the 2nd International Symposium on High Performance Computer Architecture* (*HPCA*), San Jose, CA, 1996, pp.254∼263.

[10] Collins J, Tullsen D, Wang H, Shen J. Dynamic speculative precomputation. In *Proc. the 34th International Symposium on Microarchitecture*, Austin, Texas, 2001, pp.306∼317.

[11] Wessam Hassanein, José Fortes, Rudolf Eigenmann. Data forwarding through in-memory precomputation threads. In *Proc. the International Conference on Supercomputing* (*ICS*), 2004.

[12] Hughes C J. Prefetching linked data structures in systems with merged DRAM-logic [Thesis]. University of Illinois at Urbana-Champaign, Technical Report UIUCDCS-R-2001-2221, May 2000.

[13] Liao S, Wang P, Wang H, Hoflehner G, Lavery D, Shen J. Post-pass binary adaptation tool for software-based speculative precomputation. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'02*), Berlin, Germany, 2002, pp.117∼128.

[14] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proc. the 28th Annual International Symposium on Computer Architecture*, Göeborg, Sweden, 2001, pp.40∼51.

[15] Amir Roth, Gurindar S Sohi. Speculative data-driven multithreading. In *Proc. the 7th International Symposium on High Performance Computer Architecture*, Nuevo Lenone, Mexico, 2001, p.37.

[16] Craig Zilles, Gurindar Sohi. Execution-based prediction using speculative slices. In *Proc. the 28th Annual International Symposium on Computer Architecture* (*ISCA*), Göeborg, Sweden, 2001, pp.2∼13.

[17] Yang C L, Lebeck A R. Push vs. pull: Data movement for linked data structures. In *Proc. the International Conference on Supercomputing* (*ICS*), Santa Fe, New Mexcio, 2000, pp.176∼186, pp.176∼186.

[18] James E Smith. Decoupled access/execute computer architectures. In *Proc. the 9th Annual International Symposium on Computer Architecture* (*ISCA*), Gold Coast, Queensland, 1982, pp.112∼119.

[19] Culler D, Singh J P, Gupta A. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, ISBN 1558603433, August 1998.

[20] Xian-He Sun, Surendra Byna. Data-access memory servers for multi-processor environments. IIT CS TR-2005-001, November 2005, http://www.cs.iit.edu/∼suren/research.html.

[21] Burger D C, Austin T M, Bennett S. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report 1308, University of Wisconsin-Madison Computer Sciences, 1996.

[22] Surendra Byna, Xian-He Sun, William Gropp, Rajeev Thakur. Predicting the memory-access cost based on data access patterns. In *Proc. the IEEE International Conference on Cluster Computing*, San Diego, September 2004, pp.327∼336.

[23] Annavaram M, Patel J M, Davidson E S. Data prefetching by dependence graph pre-computation. In *Proc. the 28th International Symposium on Computer Architecture* (*ISCA*), Göeborg, Sweden, 2001, pp.52∼61.

[24] Kohout N, Choi S, Kim D, Yeung D. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proc. the 10th International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001, pp.268∼279.

[25] Roth A, Moshovos A, Sohi G S. Dependence based prefetching for linked data structures. In *Proc. the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 1998, pp.115∼126.
[26] Ilya Ganusov, Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proc. the 14th Annual International Conference on Parallel Architectures and Compilation Techniques* (*PACT'05*), Saint Louis, MO, 2005, pp.350∼360.
[27] Conway J H, Guy R K. The Book of Numbers. Springer-Verlag, New York, 1996, ISBN: 038797993X.
[28] Box G E P X, Jenkins G M, Reinsel G C. Time Series Analysis: Forecasting and Control. 3rd ed, Prentice Hall, 1994.
[29] Jack Doweck. Inside Intel core microarchitecture and smart memory access. White paper, Intel Research website, Available online at http://download.intel.com/technology/architecture/sma.pdf, 2006.
[30] Sun Microsystems. UltraSPARC IV Processor Architecture Overview. www.sun.com/processors/whitepapers/us4_whitepaper.pdf
[31] IBM. Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell/.
[32] Thomas R Puzak, A Hartstein, P G Emma, V Srinivasan. When prefetching improves/degrades performance. In *Proc. the 2nd Conference on Computing Frontiers*, Ischia, Italy, May 04∼06, 2005, pp.342∼352.
[33] Standard Performance Evaluation Corporation. SPEC Benchmarks, http://www.spec.org/.
[34] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software,* 1990, 16(1): 1∼17.
[35] John D McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture,* 1995, http://www.cs.virginia.edu/stream.
[36] Sherwood T, Perelman E, Calder B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proc. the International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Spain, 2001, pp.3∼14.
[37] Dahlgren F, Dubois M, Stenström P. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems,* 1995, 6(7): 733∼746.
[38] Yue Liu, David R Kaeli. Branch-directed and stride-based data cache prefetching. In *Proc. the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, October 7∼9, 1996, pp.225∼230.
[39] Mowry T, Gupta A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing,* June 1991, 12(2): 87∼106.
[40] Pai V S, Ranganathan P, Abdel-Shafi H, Adve S. The impact of exploiting instruction-level parallelism on shared-memory multiprocessors. *IEEE Transactions on Computers,* February 1999, 48(2): 218∼226.
[41] Zhou H. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proc. the 2005 International Conference on Parallel Architectures and Compilation Techniques* (*PACT'05*), Saint Louis, MO, 2005, pp.231∼242.
[42] Solihin Y, Lee J, Torrellas J. Using a user-level memory thread for correlation prefetching. In *Proc. International Symposium on Computer Architecture*, Anchorage, Alaska, May 2002, pp.171∼182.

Xian-He Sun received his B.S. degree in mathematics in 1982 from Beijing Normal University, China, and received his M.S. degree in mathematics, M.S. and Ph.D. degrees in computer science in 1985, 1987, and 1990, respectively, all from Michigan State University, USA. He was a post-doctoral researcher at the Ames National Laboratory, USA, a staff scientist at the ICASE, NASA Langley Research Center, an ASEE fellow at the US Navy Research Laboratories, and was an associate professor in the Department of Computer Science, Louisiana State University before he joined the Computer Science Department, Illinois Institute of Technology (IIT) in August 1999. Currently he is a professor of computer science at IIT, a guest faculty in the Mathematics and Computer Science Division at the Argonne National Laboratory, and the director of the Scalable Computing Software Laboratory at IIT. He was on sabbatical leave during the 2006∼2007 academic year working in the Computing Division at the Fermi National Accelerator Laboratory as a visiting scientist. Dr. Sun's research interests include high performance computing, performance evaluation, and distributed systems. More information about Prof. Sun can be found at www.cs.iit.edu/∼scs/sun.

Surendra Byna received his B. Tech. in electronics and telecommunication engineering in 1997 from Jawaharlal Nehru Technological University, India. He received his M.S. and Ph.D. degrees in computer science in 2001 and 2006, respectively. Currently, he is a senior research associate in the Computer Science Department at Illinois Institute of Technology (IIT), Chicago. He is also a guest researcher at Argonne National Laboratory and a faculty member at Scalable Computing Software Laboratory at IIT. Dr. Byna's research interests include high performance computing, data access performance evaluation and optimization, parallel I/O, and power aware computing.

Yong Chen received his B.E. degree in computer engineering in 2000 and M.S. degree in computer science in 2003 from University of Science and Technology of China. Currently, he is pursuing his Ph.D. degree in computer science from Illinois Institute of Technology, Chicago. His current research focuses on parallel and distributed computing in general, and on performance evaluation, optimization, data access performance, and parallel I/O in particular.