Improving Data Access Performance with Server Push Architecture

Xian-He Sun, Surendra Byna, Yong Chen
Illinois Institute of Technology
Department of Computer Science
Chicago, IL 60616 USA
{sun, renbyna, chenyon1}@iit.edu

Abstract

Data prefetching, where data is fetched before CPU demands for it, has been considered as an effective solution to mask data access latency. However, the current client-initiated prefetching strategies do not work well for applications with complex, non-contiguous data access patterns. While technology advances continue to enlarge the gap between computing and data access performance, trading computing power for data access delay has become a natural choice. We propose a server-based data-push approach. In this server-push architecture, a dedicated server named Data Push Server (DPS) initiates and proactively pushes data closer to the client in time. We present the DPS architecture and study the issues such as what data to fetch, when to fetch, how to push, and data access modeling.

1. Introduction

Data access latency is a major contributor to the gap between peak performance and sustained performance of current high-end computing (HEC) machines. Performance of processors and network interconnects are increasing multiple times faster than that of memory and I/O. This causes large stall times in processors to wait for data to arrive. The data access performance must be improved to utilize the capacity of large supercomputers efficiently.

Prefetching has been considered as an effective technique for masking data access latency. Prefetching fetches data before it is requested by a processing unit. Prefetching requires an algorithm to predict future references spatially (*what* data to prefetch) and temporally (*when* to issue a prefetch). Current prefetching implementations typically predict the address of the next load address when a constant stride between successive accesses is found. However, many data access patterns are complex, which are formed by variable strides that have regularity. Various algorithms can predict these patterns, but are too complex to implement. Existing prefetchers are limited by the complexity of these prediction algorithms. They lack adaptability to choose prediction algorithms based on the history of data access patterns. Using complex algorithms takes a bite of processing power and may diminish the benefits of prefetching.

Emerging multicore processors and high-end computing machines with thousands of nodes are good candidates for utilizing part of processing power to handle the complexity of prefetching. We propose a *push-based* prefetching using a server, called 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' also means that, unlike traditional client-initiated prefetching, DPS initiates prefetching. DPS can adapt to complex prediction algorithms for more aggressive prediction and can push data into multiple *processing units*[1]. It can adaptively choose a prediction method based on the history of accesses and compiler hints. This is very beneficial to HEC, where few of the so called "grand challenge applications" often running repeatedly. We also use temporal data access information to predict *when* to push data. This avoids costly synchronization needed by pre-execution strategies [3, 4, 7, 10, 11, 13] to initiate prefetching *in time*. DPS can fit at multiple levels of memory hierarchy including to perform I/O prefetching. We enhance the SimpleScalar simulator [1] to provide performance results at cache prefetching level. While these results on benchmarks are preliminary, they show that DPS has merits and has a real potential.

2. Related Work

Data prefetching is a well studied research area in computer architecture. Sequential prediction strategies

[1] The notion of *processing unit* refers to a processing core in multi-core processors and to a computing node in SMP and cluster computers.

prefetch next *k* lines of data, while strided prediction strategies prefetch future strides based on past accesses [12]. With the increasing complexity of these methods, the benefits of prefetching diminish in the traditioanl client-initiated prefetching. Software-controlled prefetching [9] gives control to a developer or a compiler to insert prefetching instructions into programs. 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.

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 among numerous proposals using pre-execution include Luk et al.'s Software controlled pre-execution [8], Liao et al.'s Software-based speculative precomputation [7], Roth et al.'s Data-driven multithreading [10], and Hassanein et al.'s data forwarding [4]. Many of these methods often rely on compiler support to select slices of code to pre-execute and to trigger execution of that code. Zhou [13] proposed *dual-core execution* (DCE) and Ganusov et al. [3] proposed *future execution* (FE) to utilize idle cores of a CMP to speed up single threaded programs. In contrast to pre-execution approaches, our DPS resides on a dedicated data server and adaptively chooses stride prediction strategies. DPS is designed to serve multiple processing cores simultaneously, where as DCE and FE are tightly coupled to one core. In DPS, we target to predict 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. Solihin et al. [11] propose memory-side prefetching, where a memory processor is designed to reside within the main memory to observe history of L2 cache misses that pushes data into L2 cache. We use 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. DPS also predicts *when* to push data based on temporal pattern of data accesses for in-time prefetching.

3. Data Push Server

Figure 1 shows the structure of Data Push Server (DPS). 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

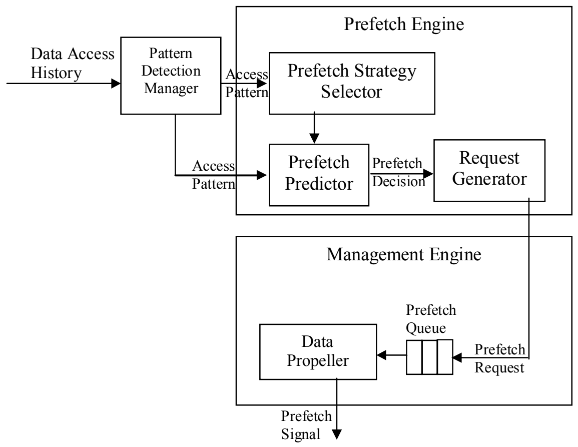

Figure 1. Components of Data Push Server

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. 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. Predicted 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.

Source and destination of DPS vary based on where it is implemented. In a multi-core processor environment, the source is its main memory, and the destination is cache memory. In I/O prefetching, the source is a disk and the destination is the main memory of a client node. Figure 2 shows a scenario of DPS system running on a computing core, serving processing cores *1,2, ..., m*. We show that each core in a multicore processor environment contains its own L1 and L2 cache memories and shares the memory among other cores. The core, on which DPS is running, observes the data access patterns of L1 cache of cores *1 to m*, and predicts the future accesses correspondingly. The data (prefetched cache line or PCL) is pushed from the shared main memory to the prefetch cache (PC) of each client core by issuing prefetch signals

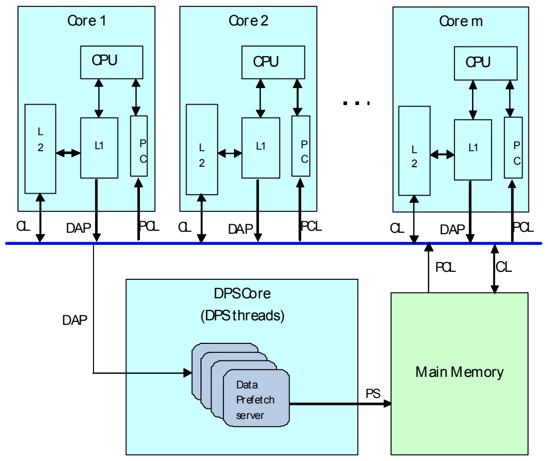

Figure 2. Data Push Server for Multi-core Processors

(PS) to the main memory. Regular memory operations related to raw cache misses caused by an application are served by main memory directly. These cache lines are read or written by L2 cache and this data (CL) is transferred between main memory and L2 cache. CPU on each core accesses both L1 cache and prefetch cache simultaneously. An L1 cache miss is propagated to lower level L2 cache. A prefetch cache miss is discarded. Similarly, DPS can be placed in computing nodes of SMPs and clusters at various levels. In SMPs, data can be prefetched from shared memory to compute nodes' local memory. Another scenario is that DPS acting as I/O prefetcher to push data from I/O servers to client nodes as a part of parallel file systems. In the following section, we discuss the functionality of DPS system components in detail.

4. Functionality of DPS

4.1 Prediction of Future Data Accesses

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 [12] that require compiler and user provided hints. 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 our 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. 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.*

The *prefetch strategy selector* (PSS) chooses a prediction strategy based on initial information regarding a pattern. Many strategies exist to predict future references with similar strides or patterns of strides [12]. However, patterns with variable strides and repetitions need more analysis to find regularity among them. With DPS, as dedicated computing power is available for prediction, we can use Markov Chain [5] and a novel Multi-level Distance Table (MLDT) [2] based predictions to find regular patterns with constant stride as well as variable stride accesses and repeating patterns.

4.2. Prediction of When to Prefetch

The issue of *when* to prefetch in existing prediction based prefetching 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 (Figure 3): 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 ($T_{pred} + T_{lat}$) = T_{Δ}, the prefetching is *in time* and is the most effective.

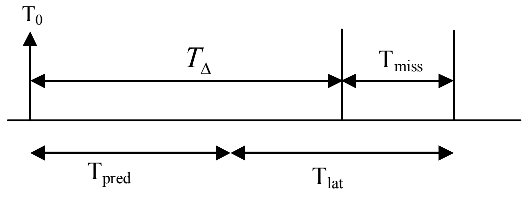

Figure 3. In time prefetching

If T_{miss} denotes the penalty caused by a cache miss, there is a partial gain of performance improvement based on how much of T_{miss} is overlapped if $(T_{pred} + T_{lat}) > T_{\Delta}$ and $(T_{pred} + T_{lat}) < (T_{\Delta} + T_{miss})$. If data is prefetched too early, i.e. $(T_{pred} + T_{lat}) < T_{\Delta}$, there is a possibility of replacing useful cache lines.

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, the *request generator* decides when to prefetch. The request generator chooses what future reference (*prefetch distance*) has to be prefetched 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 *prefetch distance* so that $(T_{pred} + T_{lat})$ is 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 [2] method to identify temporal pattern in order to predict T_{Δ}. In the future we plan to use ARIMA models [5] to predict temporal access patterns.

4.3. Pushing Predicted 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.

4.4. Suggestions for Implementation

In order to implement DPS and 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, the DPS core collects data access history of the processing cores. DPS also requires hardware support to push data from memory to upper level cache of the processing cores. 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, but not to prefetch data to other cores. Emerging chip-level multiprocessors (IBM's Cell processor, ClearSpeed's co-processors etc.), have many processing cores. These processors show some prospect to implement DPS. The cores of a Cell processor have an internal bus, which can be used for observing patterns of their local memories and for pushing data directly to their local memory. Address translation also needs some support. TLB misses may occur if address mapping is not updated at compute core. This can be solved by providing separate virtual prefetch cache. Such provision reduces false replacement of data from L1 or L2 caches. As the processing unit searches data cache and prefetch cache in parallel, the server-based data-push model benefits more by reducing data cache misses further.

5. Experimental Results

We compare the performance results for three cases: base case, strided prefetching and DPS prefetching. Base case performs no prefetching. The strided prefetching strategy predicts the next stride based on history of recent accesses and a prefetch instruction is issued on the occurrence of a cache miss. *Prefetching distance* is constant for strided prefetching. Prefetching is initiated by the DPS core for DPS strategy. Prefetching distance varies based on *request generator* decisions on *when* to prefetch.

We evaluate the performance by using an extended version of the SimpleScalar toolset v4.0 [1]. The baseline simulator configuration consists of a four-issue dynamic superscalar cores similar to that of Alpha 21264, with L1 cache (32 KB, 2-way, 64 byte cache line, and 2 cycle hit time) and L2 cache (1MB, 4-way, 64 byte line, 12 cycle hit time, and 100 cycle miss penalty) To apply strided prefetching, we modified the *sim-outorder* simulator using a 512-entry reference prediction table (RPT) [12]. The prefetch distance is constant and set as 8 for strided prefetching. Our experiments have shown that this prefetch distance has least cache misses for the tested benchmarks. To implement DPS prefetching strategy, we use a 512-entry Data Access History (DAH) [2] structure to collect load instruction information. The DAH is similar to RPT, but stores more information. DAH has a tag, count, tail and head pointer fields. 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. To simulate the DPS core, we modified the *sim-outorder* simulator to add another Alpha 21264 core that contains all the components of DPS core. Operation of this core does not affect the cycles or instructions of the processing core. To simulate data prefetching functionality, we modified the memory module of the DPS core to introduce an instruction to prefetch data into the L1 cache of processing core.

We present performance results of SPEC CPU2000 benchmarks that have poor L1 cache performance. Figure 4 shows L1 cache miss rates of these benchmarks. With DPS prefetching, L1 miss rates are reduced significantly for all the benchmarks. 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.

Figure 5 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 DPS prefetching is implemented without a dedicated DPS core, i.e. DPS prefetching is implemented on the same processing unit, 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, when it is implemented on the same processing core, the overall

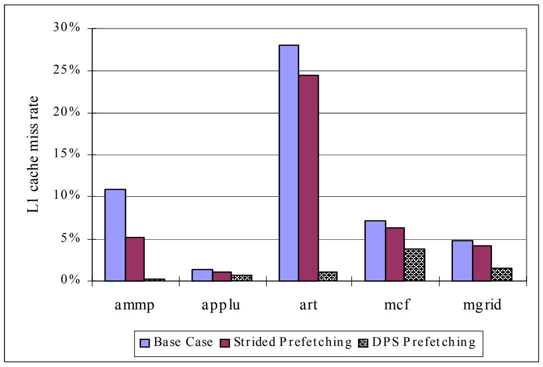

Figure 4. Comparison of L1 miss rate

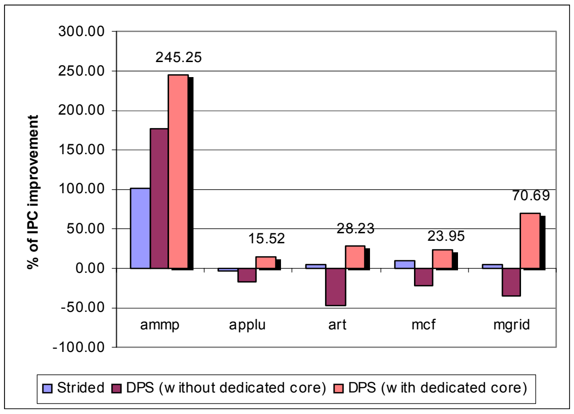

Figure 5. IPC results with DPS prefetching

performance degrades. With the use of a dedicated memory server, the IPC values improve significantly, benefiting from aggressive prefetching.

These performance results show the potential of using a dedicated DPS 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 broadens the impact of CMP architectures in bridging the divergence gap of HEC.

6. Conclusions

In this study, we have presented the server-based data push architecture, called Data Push Server (DPS), for effectively masking processor stall time. 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 multi-core processors. Initial simulation results show that DPS has a profound potential to improve the memory access performance of various data access patterns.

We have only demonstrated some potential of DPS in this study. Many research issues remain open. We plan to investigate 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 intend to explore more accurate pattern prediction algorithms, such as time series analysis models.

Acknowledgments

This research was supported in part by national science foundation under NSF grant CNS0509118 and CCF-0621435.

References

1. D.C. Burger, T.M. Austin, and S. Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set", Technical Report 1308, University of Wisconsin-Madison Computer Sciences, 1996.
2. S. Byna, X-H. Sun, Y. Chen, "Server-based Data Push for Multiprocessor Environments", IIT CS TR-2007-12, January 2007, http://www.cs.iit.edu/~suren/research.html
3. Ilya Ganusov and M. Burtscher, "Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors", in Proceedings of the 14th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT'05), 2005.
4. W. Hassanein, J. Fortes and R. Eigenmann. "Data Forwarding through In-Memory Precomputation Threads", in Proceedings of the International Conference on Supercomputing (ICS), 2004.
5. D.Joseph and D. Grunwald. "Prefetching Using Markov Predictors", in Proceedings of the 24th International Symposium on Computer Architecture, Denver-Colorado, pp 252-263, 1997.
6. N. Kohout, S. Choi, D. Kim, and D. Yeung, "Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes," in Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT-01), 2001.
7. S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen, "Post-Pass Binary Adaptation Tool for Software-Based Speculative Precomputation", in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02), 2002.
8. Chi-Keung Luk, "Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors", in Proceedings of the 28th International Symposium on Computer Architecture (ISCA), pages 40-51, 2001.
9. T. Mowry and A. Gupta, "Tolerating Latency Through Software-controlled Prefetching in Shared-memory Multiprocessors," Journal of Parallel and Distributed Computing, Volume 12, Issue 2, June 1991.
10. Amir Roth and Gurindar S. Sohi, "Speculative data-driven multithreading", in Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA), 2001.
11. Y.Solihin, J.Lee, and J.Torrellas "Using a User-Level Memory Thread for Correlation Prefetching", in Proceedings of the International Symposium on Computer Architecture (ISCA), May 2002, 171-182.
12. Steven P. Vanderwiel , David J. Lilja, "Data prefetch mechanisms", ACM Computing Surveys (CSUR), v.32 n.2, p.174-199, June 2000
13. H. Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window", in Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques (PACT'05), 2005.