A Taxonomy of Data Prefetching Mechanisms

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

Abstract

Data prefetching has been considered an effective way to mask data access latency 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 proposed in the last few years to reduce data access latency by taking advantage of multi-core architectures. In this paper, we propose a taxonomy that classifies various design concerns in developing a prefetching strategy. We discuss various prefetching strategies and issues that have to be considered in designing a prefetching strategy for multi-core processors.

1. Introduction

Data prefetching is a data access latency hiding technique, which decouples and overlaps data transfers and computation. In order to reduce CPU stalling on a cache miss, data prefetching predicts future data accesses, initiates a data fetch, and brings the data closer to the computing processor before it is requested.

A data prefetching strategy has to consider various issues in order to mask data access latency efficiently. It should be able to predict future accesses accurately and to move the predicted data from its source to destination *in time*. Several proposed strategies predict future data accesses using recent history of data accesses from which patterns can be recognized [4][5] [8][10], using compiler and user provided hints [14][18], analyzing traces of past execution of applications or loops [12], and running a helper thread ahead of actual execution of an application to predict cache misses [1][11][20][21][27]. Among these strategies, predicting future data accesses based on recent history has been popular and partially implemented at hardware level in existing processors. Compiler and user provided hints are used in software-level prefetching [13][16]. Helper-thread initiated prefetching is becoming popular in multi-threaded and multi-core processors. Prefetching data exactly *in time* is a challenging task. Data should not be prefetched too early or too late. In addition to considering *what to prefetch* and *when to prefetch* aspects, complexity of executing prefetching methods should be low in order not to block actual processing.

Figure 1 shows various scenarios of implementing prefetching strategies. In Scenario *A*, a prefetch engine (PE) observes history of L1 cache misses and initiates prefetch operations. In multi-threaded and multi-core processors, pre-execution based approaches use a separate thread to predict future accesses (scenario *B*). A prefetching-thread pre-executes data references of a main computation-thread and initiates prefetching data into a shared cache memory (L2 cache) earlier than the computation-thread. In memory-side prefetching strategy, (scenario *C*) the prefetching-thread is executed on an intelligent main memory, where a memory processor pre-executes helper-threads. The predicted data is pushed towards the processor. From

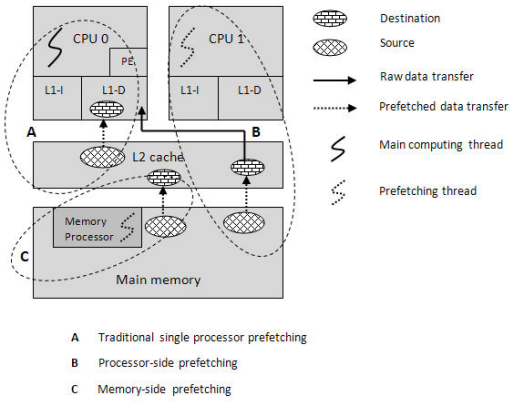

Figure 1. Prefetching scenarios

these scenarios, it is evident that in addition to predicting *what* and *when* to prefetch, sources, destinations, and initiators of prefetching play primary role in designing an effective prefetching strategy.

In this paper, we provide a taxonomy of prefetching strategies that primarily captures design issues of prefetching strategies. VanderWiel et al. [24] and Oren [17] discussed the definitions of prefetching and compared various prefetching strategies in the context of single-core processors in their surveys. Their survey provides a taxonomy addressing *what*, *when*, and *where* (destination of prefetching) questions for hardware prefetching and software prefetching. The emergence of multi-thread and multi-core processor architectures brought new opportunities and challenges in designing effective prefetching strategies. We propose a taxonomy of prefetching mechanisms based on a comprehensive study of hardware prefetching, software prefetching, prediction and pre-execution based prefetching, and more importantly, strategies that are novel to multi-core processors.

The rest of the paper is organized as following: Section 2 presents the taxonomy to classify data prefetching strategies. We briefly discuss the prospects and challenges of prefetching on multi-core processors in Section 3 and conclude our discussion in Section 4.

2. Taxonomy

We take a top-down approach to characterize and classify the design issues of prefetching strategies. Figure 2 shows the top layer of our classification, 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.

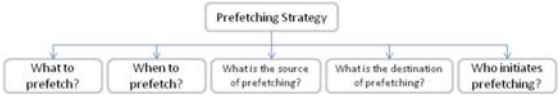

Figure 2. Five fundamental issues of prefetching

2.1. What to prefetch?

Predicting *what* data to prefetch is the most important requirement of prefetching. If a prefetching strategy can predict the occurrence of future misses, then prefetch instructions can be issued ahead and bring that data by the time the cache misses.

To mask the stall time caused by cache misses effectively, the accuracy of predicting *what to prefetch* must be high. Predicting future data references accurately is critical. Low accuracy leads to cache pollution. Figure 3 shows a classification of predicting *what* data to prefetch based on where it is implemented and on various techniques.

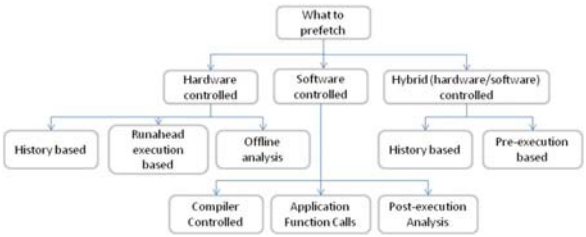

Figure 3. Predicting *what* data to prefetch

2.1.1. Hardware controlled strategies. In *hardware controlled* data prefetching, prefetching is implemented in hardware. Various methods support hardware controlled prefetching.

History-based prediction is the most commonly used among hardware controlled data prefetching strategies. In these strategies, a prefetch engine (PE) is used to predict future data references and to issue prefetching instructions. All the components of prefetching are implemented within a processor and they do not require any user interference. PE observes the history of data accesses or the history of cache misses to predict future accesses by a processor. We discuss various algorithms used in history based prediction in Section 2.1.4.

Runahead execution uses idle cycles or cores to run instructions while a CPU is stalled or idle. Zhou [26] proposed *dual-core execution* (DCE) approach uses an idle core of a dual-core processor to construct large, distributed instruction window and Ganusov et al.'s [6] *future execution* (FE) uses an idle core to pre-execute future loop iterations using value prediction.

Off-line analysis strategy is another hardware controlled prefetching approach. Kim et al. [12] proposed a method, where data access patterns are analyzed for hotspots of code that are frequently executed. This approach works well for applications that refer to similar data access pattern.

2.1.2. Software controlled strategies. *Software-controlled* prefetching [13][16] gives control to a developer or to a compiler to insert prefetching instructions into programs. From Figure 3, software controlled prefetching can use *compiler controlled* prefetching instructions or *function calls* in source code or prefetching instructions inserted based on *post-execution analysis*. Many processors provide support for such prefetching instructions in their instruction set. Compilers or application developer can insert these prefetch instructions or built-in routines provided by

compilers. Software-controlled prefetching puts burden on developers and compilers, and is less effective in overlapping memory access stall time on ILP processors due to late prefetches and resource contention [20]. *Post-execution analysis* can also be used, where traces of data accesses are analyzed for patterns.

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 access patterns. These methods require support from hardware to run helper-threads that are specifically executed to prefetch data. They require software support to synchronize with the actual computation thread. The helper-thread based prefetching strategies either analyze history of data accesses of a computation thread or pre-execute data intensive parts of the computation thread that warms up a shared cache memory by the time a raw cache miss occurs.

History based hybrid prediction strategies (Solihin et al. [20]) analyze history of accesses to predict future accesses and prefetch data. *Pre-execution based* methods [15][19][1][7] using a helper-thread to execute slices of code ahead of computation thread.

2.1.4. History-based prediction algorithms. From Figure 3, hardware controlled, software controlled, and hardware/software controlled approaches use prediction algorithms based on history of data accesses or cache misses. Prediction algorithms search for various patterns among history of data accesses. Figure 4 shows a classification of data access patterns based on spatial distance between accesses, their repeating behavior and request size of accesses. Spatial patterns are divided based on the number of bytes (also called as *stride*) between successive accesses as contiguous, non-contiguous, and combinations of both. Non-contiguous patterns are further classified by the property of strides between accesses. Data access patterns repeat when loops or functions execute repeatedly. We classify these patterns as either single occurrence or repeating patterns. Request size in each access may be fixed or variable. This classification captures a wide range of data accesses.

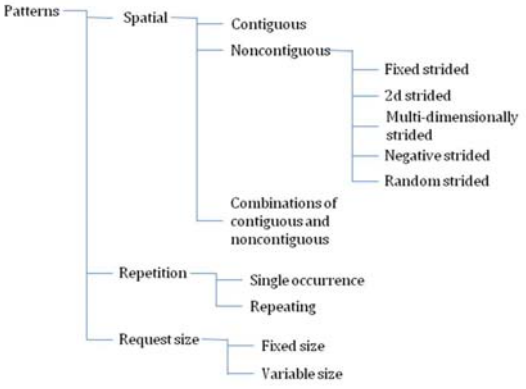

Figure 4. Classification of data access patterns

Several prediction algorithms have been proposed to find these patterns. Sequential prefetching [5] fetches contiguous cache blocks by taking advantage of locality. Stride prefetching approach [4] predicts future accesses based on strides of the recent history. Strided prefetching strategies maintain a reference prediction table (RPT) to keep track of recent data accesses. To capture repetitiveness of data accesses, Markov prefetching [8] was proposed. Distance prefetching [10] uses Markov chains to build and maintain probability transition diagram of strides among data accesses. Multi-level Difference Table (MLDT) [22] uses time-series analysis method to predict future accesses in a sequence, by finding the differences in a sequence to multiple levels.

2.2. When to prefetch?

The time to issue a prefetch instruction has significant effect on the overall performance of prefetching. Prefetched data should arrive 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 prefetching data) and the time for the occurance of next cache miss. If the total prefetching overhead exceeds the time of next cache miss, adjusting prefetching distance can avoid late prefetches. Figure 5 shows a classification of various methods used in deciding when to prefetch.

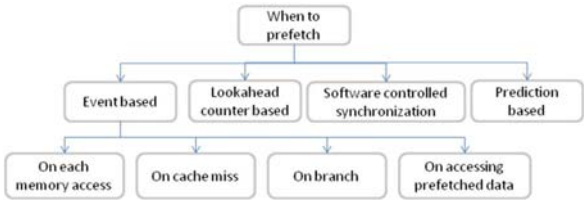

Figure 5. Predicting *when* to prefetch

Event based mechanism issues a prefetch instruction on 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*. Prefetch on miss is a common implementaion on existing processors as it is simple to implement.

Tagged prefetching [3] initiates a prefetch instruction when a data access hits previously prefetched data block for the first time. Branch directed prefetching [3] suggests that since branch instructions determine which instruction path is followed, data access patterns are also dependent upon branch instructions.

Chen et al. [4] have proposed using a *lookahead program counter* (LA-PC). In loop codes, 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 prefetching functions early enough to prefetch data. Mowry et al. [16] provide an algorithm to calculate prefetching distance [13]. According to this algorithm, prefetching instructions are called strictly for data references that would cause cache misses. In helper-thread based approaches, periodic synchronization of computation thread with helper-thread is required to prevent late prefetches or very early prefetches. Kim et. al. [11] and Song et. al. [21] use synchronization to prevent helper-thread execution lagging behind computation thread.

In many applications data access bursts follow certain pattern. By analyzing the time intervals, future data bursts can be predicted to start prefetching. Server-based push prefetching [22] uses *prediction based* strategy to analyze when to prefetch.

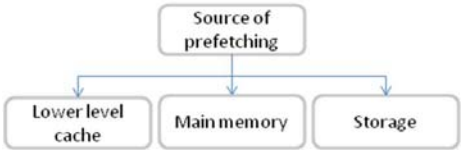

Figure 6. Source of prefetching

2.3. What is the 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. 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 know where the latest copy of data is. In existing deep memory hierarchies with write back policy, data can exist at any level of memory hierarchy. In single-core processors, prefetching source is usually the main memory or 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. In this paper, the focus of our discussion is limited to cache and memory level prefetching.

2.4. What is the destination of prefetching?

Destination of prefetched data is another major concern of prefetching strategy. Prefetching destination should be closer to CPU than a prefetching source in order to obtain performance benefits. As shown in Figure 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 processor core or shared by multiple cores.

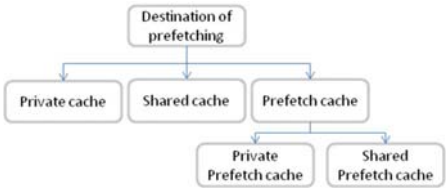

Figure 7. Destination of prefetching

While the best destination of prefetching data is the private cache to avoid cache pollution, there are various design issues that affect such prefetch strategy. One of them is the small size of cache memory. Prefetching data into this cache may cause cache pollution. To reduce the cache pollution, a dedicated buffer called prefetch cache [9] was proposed. In multi-core 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 (e.g. helper-thread based pre-execution). Casmira et al. [2] proposed a Prefetch Buffer Filter (PBF).

2.5. Who initiates prefetch instructions?

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

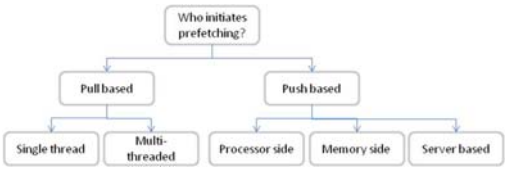

Figure 8. Initiator of prefetching

Pull based prefetching has been a common approach of prefetching in single-core processors. In this method, prefetching mechanism (prediction and initiation) is within the processor. *Multi-threaded* processors enable decoupling of data access from computing. Helper-thread based prefetching [15][19] are a couple of representative helper-thread based prefetching strategies that pull 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 [6][26] and helper-thread based prefetching methods [15][19] also can be run on a separate core on *processor side* to push data into a shared cache, which is used by the computation core.

Memory-side prefetching is relatively a new idea, where a processor residing in the main memory pushes predicted data closer to the processor [25]. *Server-based* push strategy pushes data from its source to destination without waiting for requests from processor side. Data Push Server (DPS) [22] uses a dedicated server to initiate and proactively pushes data closer to the client *in time*.

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

3. Prefetching for Multicore Processors

Designing prefetching strategies for multi-core processors poses new challenges. These challenges include multiple computing cores' competing to fetch regular data and prefetched data, while sharing memory bandwidth. With single-core processors, main memory accepts prefetching requests just from one core. In multi-core processors, prefetching requests from multiple cores may put more pressure on main memory in addition to regular data fetch requests. For example, the memory processor based solutions [7][20] are not scalable to monitor data access history or pre-execute threads and predict future references for multiple cores. This problem can be solved by decoupling data access from computing cores. In Server-based push prefetching [23], we propose using a dedicated server core to provide data access support by predicting and prefetching data for computing cores.

Another challenge of multi-core processor prefetching is cache coherence. Multi-core processors access the main memory, which is shared by multiple cores and hence at some level in the memory hierarchy they have to resolve conflicting accesses to memory. Cache coherence in multi-core processors is dealt either by directory-based approach or by snooping cache accesses. Prefetching requests to shared data can be dropped to reduce complexity of 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 available, transferring complexity to idle or dedicated cores using Server-based push prefetching architecture [23] is beneficial.

4. Conclusions

Performance gains of prefetching strategies depend on various criteria. With the emergence of multi-core and multi-threaded processors, new challenges and issues need to be considered to prefetch data. In this paper, we provide a taxonomy of the five primary issues (*what, when, destination, source, and initiator*), that are necessary in designing prefetching strategies. We discuss each issue in detail, which defines the design of a prefetching strategy using examples of various prefetching strategies. We also discuss challenges of prefetching strategies in multi-core processors. To be effective, a prefetching strategy for multi-core 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. Our server-based push prefetching selects prediction strategies considering these challenges.

Acknowledgements

This research was supported in part by National Science Foundation under NSF grant EIA-0224377 and CCF-0621435.

References

[1] M. Annavaram, J. M. Patel and E. S. Davidson, "Data Prefetching by Dependence Graph Precomputation", *in the Proceedings of the 28th International Symposium on Computer Architecture*, pp. 52-61, 2001.

[2] J. P. Casmira and D. R Kaeli, "Modeling Cache Pollution", *International Journal of Modeling and Simulation*, 19(2), pp. 132-138, 1998.

[3] Y. Liu, M. Dimitri, and D. R. Kaeli, "Branch-directed and pointer-based data cache prefetching", *Journal of Systems Architecture: the EUROMICRO Journal*, 45(12-13), pp. 1047-1073, 1999.

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

[5] F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proceedings of International Conference on Parallel Processing*, pp. 156-163, 1993.

[6] I. Ganusov and M. Burtscher, "Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors", *Proceedings of the 14th Parallel Architectures and Compilation Techniques*, 2005.

[7] W. Hassanein, J. Fortes, and R. Eigenmann, "Data Forwarding through In-Memory Precomputation Threads", *Proceedings of the 18th International Conference on Supercomputing*, 2004.

[8] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors", *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 252-263, 1997.

[9] N.P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364-373, 1990.

[10] G. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study", *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[11] D. Kim et al., "Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors", *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[12] J. Kim, K. V. Palem and W.F. Wong, "A Framework for Data Prefetching using Off-line Training of Markovian Predictors", *Proceedings of the 20th International Conference on Computer Design*, 2002.

[13] A. C. Klaiber and H. M. Levy, "An Architecture for Software-controlled Data Prefetching", *Proceedings of the 18th International Symposium on Computer Architecture*, pp.43-53, 1991.

[14] C.K. Luk and T.C. Mowry, "Compiler-based Prefetching for Recursive Data Structures", *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[15] C.K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[16] T. Mowry and A. Gupta, "Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors", *Journal of Parallel and Distributed Computing*, 12(2), pp.87-106, 1991.

[17] N. Oren, "A Survey of Prefetching Techniques", TR CS-2000-10, University of the Witwatersrand, 2000.

[18] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.- F. Wong, "Compiler Orchestrated Pre-fetching via Speculation and Predication", *Proceedings of the 11th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 189-198, 2004.

[19] A. Roth, A. Moshovos and G. S. Sohi, "Dependence Based Prefetching for Linked Data Structures," *Proceedings of the,8th International Conference on Architecture Support fo Programming Languages and Operating Systems*, pp. 115-126, 1998.

[20] Y. Solihin, J. Lee and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching", *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 171-182, 2002.

[21] Y. Song, S Kalogeropulos and P Tirumalai, "Design and Implementation of A Compiler Framework for Helper Threading on Multi-Core Processors", *Proceedings of the 14th Parallel Architectures and Compilation Techniques*, pp. 99-109, 2005.

[22] Xian-He Sun, Surendra Byna, and Yong Chen, "Server-based Data Push Architecture for Multi-processor Environments", Journal of Computer Science and Technology (JCST), 22(5), pp. 641-652, 2007.

[23] X.H. Sun, S. Byna and Y. Chen, "Improving Data Access Performance with Server Push Architecture", *Proceedings of the NSF Next Generation Software Program Workshop (with IPDPS '07)*, 2007.

[24] S. VanderWiel and D.J. Lilja, "Data Prefetch Mechanisms", *ACM Computing Surveys*, 32(2), 2000.

[25] C.L. Yang, A.R. Lebeck, H.W. Tseng and C. Lee, "Tolerating Memory Latency through Push Prefetching for Pointer-Intensive Applications", *ACM Transactions on Architecture and Code Optimization*, 1(4), pp. 445-475, 2004.

[26] H. Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window", *Proceedings of the 14th Parallel Architectures and Compilation Techniques*, 2005.

[27] C. Zilles and G. Sohi, "Execution-based Prediction Using Speculative Slices", *in Proceedings of the 28th International Symposium on Computer Architecture*, 2001.