CS 838 – Chip Multiprocessor Prefetching

Kyle Nesbit and Nick Lindberg
Department of Electrical and Computer Engineering
University of Wisconsin – Madison

1. Introduction

Over the past two decades, advances in semiconductor process technology and microarchitecture have led to significant reduction in processor clock periods. Meanwhile, advances in memory technology have led to ever increasing memory densities, but relatively minor reductions in memory access time. Consequently, memory latencies measured in processor clock cycles are continually increasing and are now on the order of hundreds of clock cycles in duration.

Cache memories help bridge the processor-memory latency gap, but, of course, caches are not always effective. Cache misses to main memory still occur, and when they do, the penalty is very high. Probably the most basic technique for enhancing cache performance is to incorporate prefetching. As the processor-memory latency gap continues to increase, there is a need for continued development and refinement of prefetch methods. Most existing prefetching research has focuses on uniprocessor prefetching. In this paper, we investigate cache prefetching, aimed specifically at prefetching in a Chip Multiprocessor (CMP).

Prefetching in a CMP system has very different constraints than uniprocessor prefetching. In a CMP, pin bandwidth and the number of transaction buffer entries (TBEs, the maximum number of outstanding memory requests) are much more important. Multiple processors are competing for off-chip bandwidth and TBEs, reducing the systems tolerance to inaccurate prefetches, where *prefetch accuracy* is the percent of prefetches that are accessed by demand fetches before they are evicted from the cache. Inaccurate prefetches waste system resources, increase bus contention, and can degrade overall system performance. Furthermore, in a directory-based system with multiple CMPs, memory latency is extremely important. Often these systems store the directory in memory, which may require a memory access to retrieve, which effectively

doubles the latency of the request (assuming memory access times is much larger than the bus transaction time [5]).

The CMP prefetching method we study is based on "stride stream buffer prefetching concentration zones" (CZones) [14]. This method, as originally proposed, divides memory into fixed size zones and looks for stride patterns in sequences of cache misses directed toward the individual zones. When it finds a stride pattern, it launches prefetch requests. This method has the desirable property of not needing the program counter values of the load instructions that cause misses, which may not be readily available at lower levels of the memory hierarchy.

Throughout the rest of this paper we support using CZone prefetching in a CMP system. In section 2, we describe related prefetching research. In section 3, we describe the implementation details of the CZone prefetching method in a CMP system. In section 4, we illustrate the advantages of CZone prefetching in a CMP. Lastly in section 5, we conclude and describe unfinished and future work.

To simplify the discussion, we assume a two level cache hierarchy throughout the paper. Of course, the proposed prefetch method can also be applied at lower cache levels if they are implemented. Hence, we when we refer to the "L2 cache", we are referring to the lowest level of the cache hierarchy, whatever it happens to be.

2. Related Work

2.1. Stride Prefetching

Stride prefetching techniques detect sequences of addresses that differ by a constant value, and launch prefetch requests that continue the stride pattern [4,9,11,16,18]. The simplest methods prefetch only unit strides, i.e. where addresses are one word apart. Some early methods indiscriminately prefetch sequential lines, e.g. sequential prefetching [16], while other methods wait for a sequential access stream to be detected before prefetching [18].

More advanced stride prefetching methods can prefetch non-unit strides by storing stride related information in a history table [4,11,14,]. A *key* (typically the program counter) indexes into the table. Each table entry 1) holds the most recent stride (the

difference between the two most recent preceding addresses) 2) the most recent address (to allow computation of the next stride), and 3) other state information that determines conditions under which a prefetch should be triggered. When the current address is *a* and a prefetch is triggered, addresses *a+s, a+2s, . . ., a+ds* are prefetched – where *s* is the detected stride and *d* is the *prefetch degree*; more aggressive prefetch implementations will use a higher value for *d*. *Arbitrary Stride Prefetching* [4] was one of the first schemes for stride prefetching. Arbitrary Stride Prefetching uses the program counter as a table index and detects load addresses with any constant stride.

Stride Stream Buffer CZone Prefetching [14] is a stride prefetching methods that does not use program counter values for table indexing. CZone prefetching was proposed for use in off-chip L2 caches attached to microprocessors where the program counter is not externally available. Instead of using the program counter, CZone prefetching partitions the memory address space into fixed-size CZones. Two memory references are in the same CZone if their high-order *n*-bits, *the CZone tag*, are the same. The value of *n* is an implementation-specific parameter. CZone prefetching uses the CZone tag to access a *filter table* for detecting constant strides among addresses within each CZone. Figure 1.

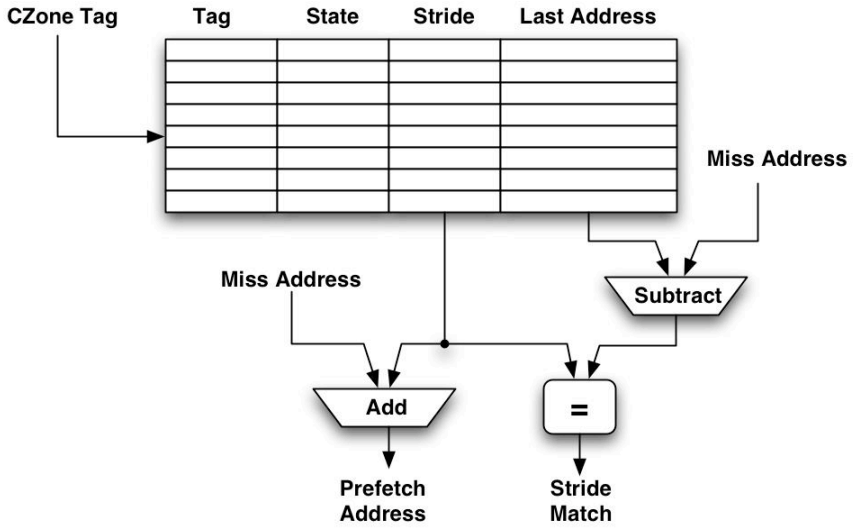


Figure 1: CZone Filter Table

2.2. Correlation Prefetching

Correlation Prefetching methods look for address sequence patterns (beyond simple strides) in order to predict future cache behavior. Most proposed correlation prefetching

methods do not use load instruction program counter values to localize address streams. *Markov Prefetching* [8] correlates global miss addresses. *Distance Prefetching* [10] was originally proposed to prefetch TLB entries, but was adapted in [13] to prefetch cache lines. The adaptation correlates *deltas* (differences in consecutive addresses) in the global miss address stream. *Tag Correlation Prefetching* [7] is a two-level correlation prefetching method that also uses the global miss address stream. The conventional cache index accesses a first level tag history table (THT). The THT contains the last *n* tags with the same cache index. These tags are combined to access a second level Pattern History Table (PHT). The PHT holds the next predicted tag, which is combined with the cache index to generate a prefetch.

3. CMP Prefetching Implementation Details

For the purposes of the this paper, we focus on stride prefetching methods, but feel that correlation prefetching in a CMP may be a good way to prefetch the hard to predict abstract access patterns (we leave this topic for future research). In general, stride prefetching methods outperform correlation prefetching methods on most workloads and require a small amount history (i.e. 4 KB) when compared with correlation prefetching methods; correlation methods have been proposed with multi-megabyte history tables. However, correlation prefetching methods have the advantage of only using the miss address stream (i.e. does not require program counter values), allowing correlation prefetching methods to be implemented anywhere in the chip's cache hierarchy.

Prefetching the lowest miss address stream in the cache hierarchy (in our case the L2) has many advantages, particularly in a CMP system. First, in a CMP, the L2 cache is often shared by all processors on the chip. Consequently, prefetching the L2 miss address stream can share prefetch history among the processors, resulting in larger history tables. Second, prefetching L2 miss addresses reduces contention on the cache ports, which is becoming increasingly important as the number of processors per chip grows. Before a prefetch is sent to the memory subsystem, it must access the L2 directory. Since the L2 miss address stream has the fewest memory references it will generate less prefetches and access the cache ports less often. Last, prefetching into the

L1 is relatively insignificant, since modern out-of-order processors can tolerate most L1 data cache misses with relatively little performance degradation.

Of the existing stride prefetching methods, we chose to study CZone prefetching. CZone prefetching has the performance of stride prefetching and is the only stride prefetching method uses just the miss address stream. Instead of prefetching into stream buffers, our implementation of CZone Prefetching, like other recent uniprocessor prefetch research [refs], prefetches into the lowest level cache (in our case the L2, see Figure 2). To keep prefetched (but not yet accessed) lines from modifying the "natural" L2 demand miss address stream, we added a new *shared prefetched* (SP) state (and an intermediate shared prefetched state). When a cache access hits a line in the shared prefetched state, the access's memory address is sent to update the prefetch structures as if it were an L2 cache miss.

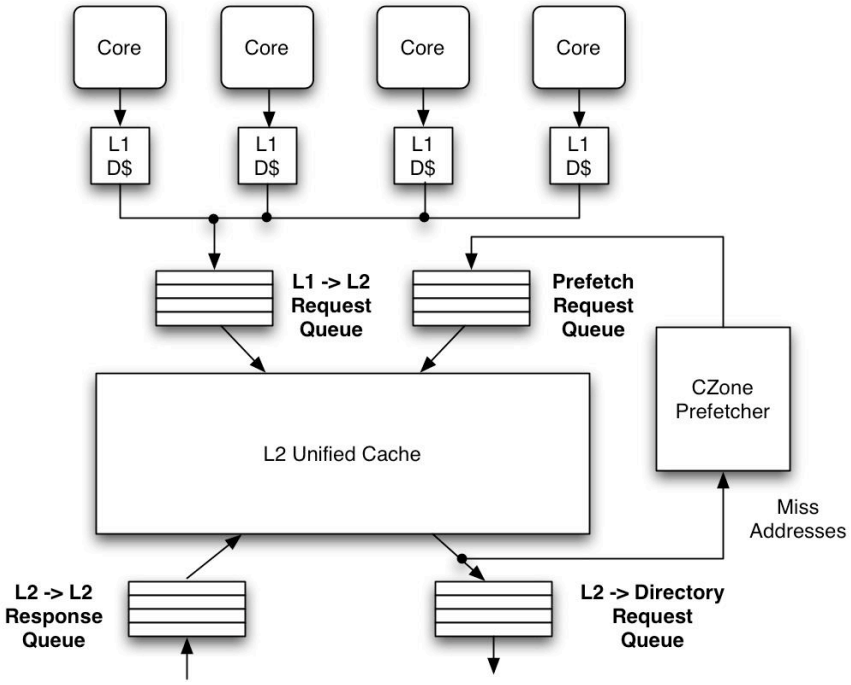


Figure 2: Prefetching Implementation

Prefetching in a CMP is more difficult than in a uniprocessor system. In addition to limited bandwidth and increased latency (as described earlier), cache coherency protocols play an important role in CMP prefetching. In this respect, our prefetching

implementation is conservative. Our prefetching implementation prefetches L1 *GET Shared* (GETS) requests (loads or instruction fetches) that miss the L2 cache and ignores *GET Exclusive* (GETX) requests. Preliminary results showed that prefetching GETX requests was too difficult to do effectively (at least with the amount of time we had), and usually resulted in high link utilization, and reduced overall performance.

Another problem that our implementation addresses is clustered cache misses, a phenomena widely known and well studied [7,12]. When prefetching the miss address stream, clustered cache misses result in clustered prefetches. These clusters of memory requests clog memory system resources, causing all processors on the chip to stall until there are available *transaction buffer entries* (TBEs). Moreover, in a CMP, clusters of cache misses from different processors can overlap, further exacerbating the problem. To prevent this scenario, our prefetching implementation tracks the number of outstanding memory requests via TBEs, and issues off-chip prefetch requests only when more than half of the TBEs are available. We found that this approach minimizes the resource contention between demand fetches and prefetches that have already been issued and are waiting for data.

In general, the number of TBEs in the L2 Cache will be an important parameter for future CMP implementations. With current trends, the number of TBEs will need to increase drastically. TBEs will need to simultaneously increase in proportion to the number of processors per chip and the number of possible outstanding memory requests. Most likely, the number of processors per will increase with Moore's law (exponentially), and the number of outstanding memory request will also increase in the future as memory latencies (measured in processor cycles) increase and as memory becomes more pipelined; i.e. RAMBUS is the beginning of this trend. Unfortunately, having a large number of TBEs is undesirable. The address field of the TBE structure must be implemented as some type of *content addressable memory* (CAM), and in general, large CAM structures are hard to design and consume a lot of power.

4. Simulation Methodology

To evaluate CMP prefetching, we use the Multifacet simulation infrastructure to simulate a multiprocessor server running scientific and commercial workloads. Our target

system is a 16 processor system (4 CMP each with 4 processors), running Solaris v9. Each processor has its own L1 Data and Instruction caches. The four processors on a chip share a L2 cache (see Figure 2). Each chip has a point to point link with all other chips in the system, a coherence protocol controller, and a memory controller for its part of the globally shared memory. The system implements sequential consistency using directory-based cache coherence. The systems configuration is shown in Table 1. We simulate different prefetch degrees to illustrate how the system is affected by an increased demand for cache ports, TBEs, and network bandwidth.

Table 1: CMP Baseline simulator configuration

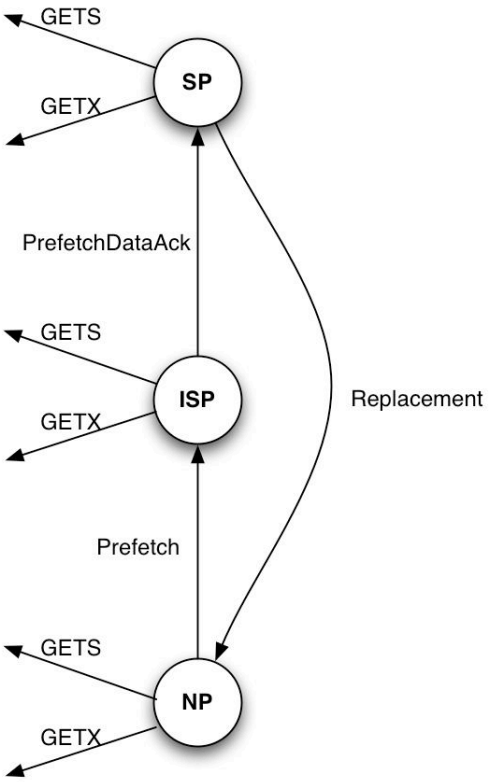
Total Number of Processors	16 processors
Processors Per Chip	4 processors/chip
L1 Instruction Cache (Per Processor)	64KB, 2-way set associative, 64 byte lines
L1 Data Cache (Per Processor)	64KB, 2-way set associative, 64 byte lines
L2 Cache (Per Chip)	4MB, 4-way set associative, 64 byte lines
L2 TBEs	64 entries
Directory Latency	200 cycles
Memory Latency	200 cycles

Our benchmarks consist of three commercial workloads and two scientific workloads. The commercial workloads consist of 100 transactions of an online transaction processing workload (OLTP), 20000 transactions of a static web serving workload (Apache), and 20000 transactions of a Java middleware workload (SPECjbb) [1], while the scientific workloads consist of a discrete grid simulation of ocean currents (Ocean) and a n-body interaction simulation (Barnes-Hut).

4.1. Results

In this section we present prefetching performance results. Based on state transition statistics collected from simulations we were able to approximate two basic prefetching metrics: 1) *Accuracy* is the percent of prefetches that are accessed by demand fetches before they are evicted from the cache and 2) *coverage* is the percent of demand fetches that are prefetched instead. Figure 3 illustrates the prefetch state transitions and the formulas used to calculate these metrics. In this diagram *SP* is the shared prefetch state, *ISP* is an intermediate state for a prefetch that is waiting for its data, and *NP* is the not present state. The actions on the arcs are all local actions, that is they were generated by

events within the chip. They are *GETS* (describe earlier), *GETX* (described earlier), *Replacement* (causes the cache line to be evicted), *Prefetch* (sends a prefetch to the appropriate directory), and *PrefetchDataAck* (an acknowledgement that the prefetch data has returned.) The notation used in the formulas is <State>Action, e.g. <SP>GETS is a prefetch hit. For this work, we only use the GETS to calculate our prefetch metrics since they are the only ones sent to the prefetcher. Figures 4 and 5 show the results of the prefetch metric calculations (using the arithmetic mean of the benchmarks).



Accuracy = (<ISP>GETS + <SP> GETS) / (<NP>Prefetch)

Coverage = (<ISP>GETS + <SP> GETS) / (<ISP>GETS + <SP> GETS + <NP>GETS)

Figure 3: Transition diagram and formulas for prefetch metrics

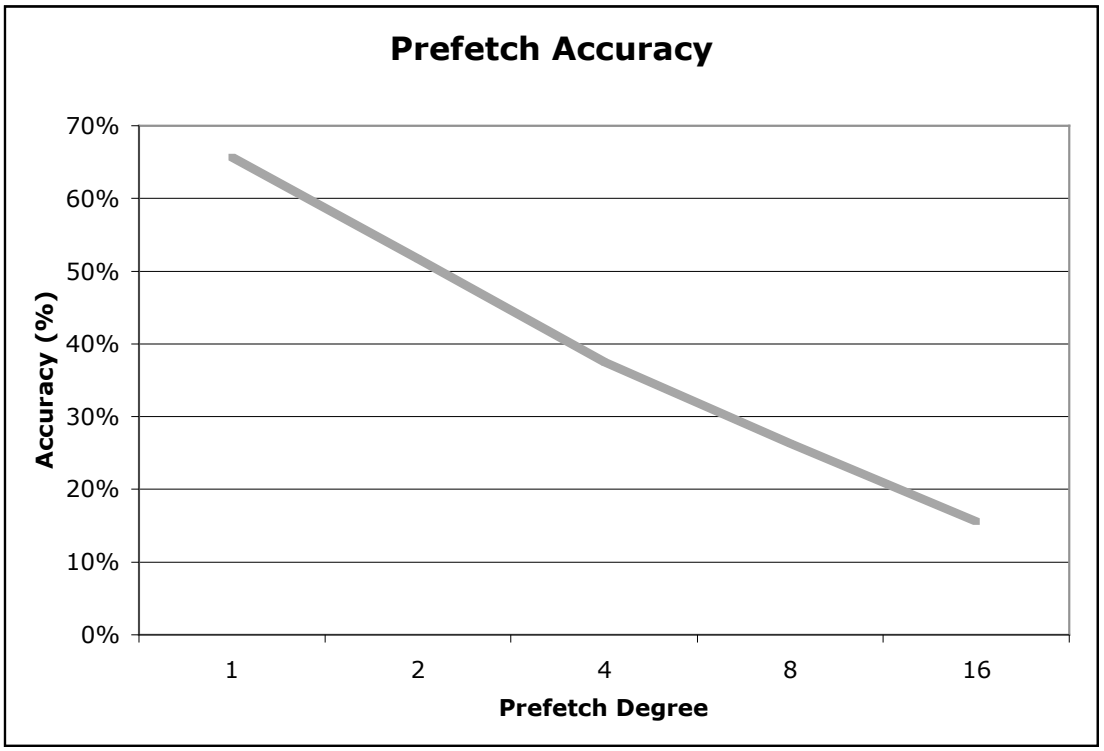


Figure 4: Prefetch Accuracy

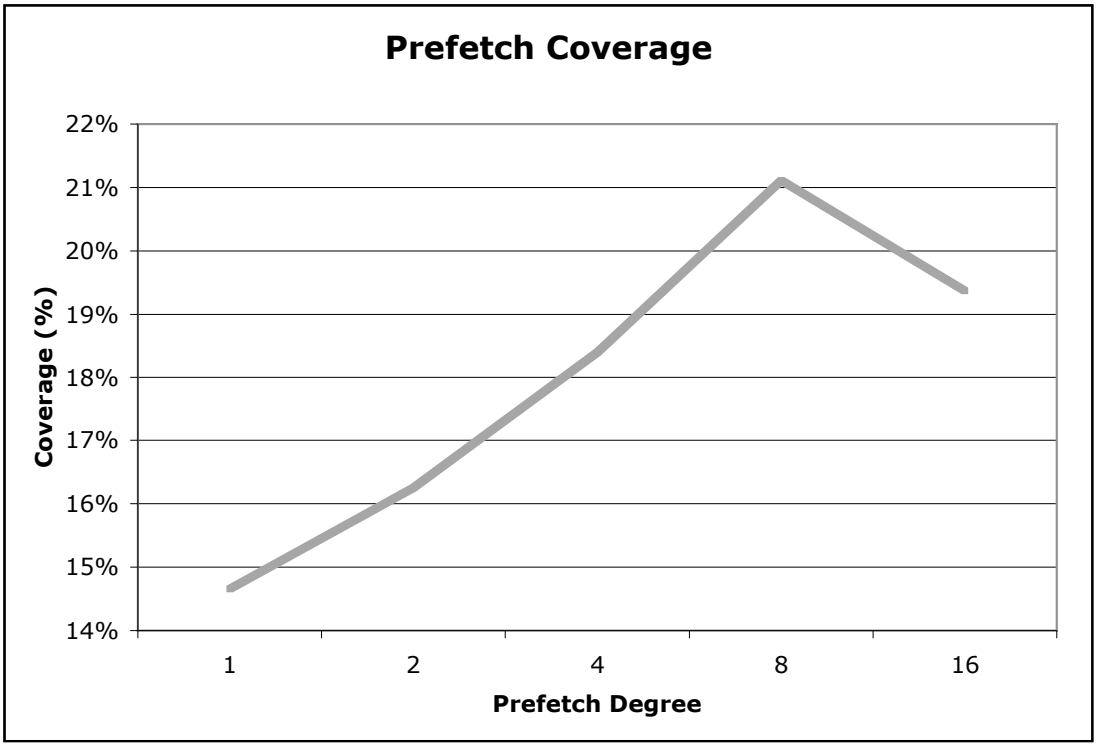


Figure 5: Prefetch Coverage

Our prefetch accuracy results are as expected: the accuracy decreases as the prefetch degree increases. The coverage results are also easy to understand. The coverage increases as prefetch degree increases, but only up to a certain point (i.e. a prefetch degree of 8). At a prefetch degree of 8, the TBEs become saturated and a large number of prefetches are dropped, and the prefetches that are not dropped have very low accuracy.

Prefetch metrics are useful to illustrate prefetch behavior and trends, but say very little about the actual performance speedup of prefetching. In Figure 6, we measure the speedup (harmonic mean of all benchmarks) vs. prefetch degree. The prefetching result of the 16 processor CMP system is compared with a 4 processor (1 processor per chip) MP system. Speedups are calculated using their respective non-prefetching (prefetch degree 0) configuration as a baseline (i.e. 16 processor configuration or 4 processor configuration). Besides having only one processor per chip, the MP system has an identical configuration (See Table 1).

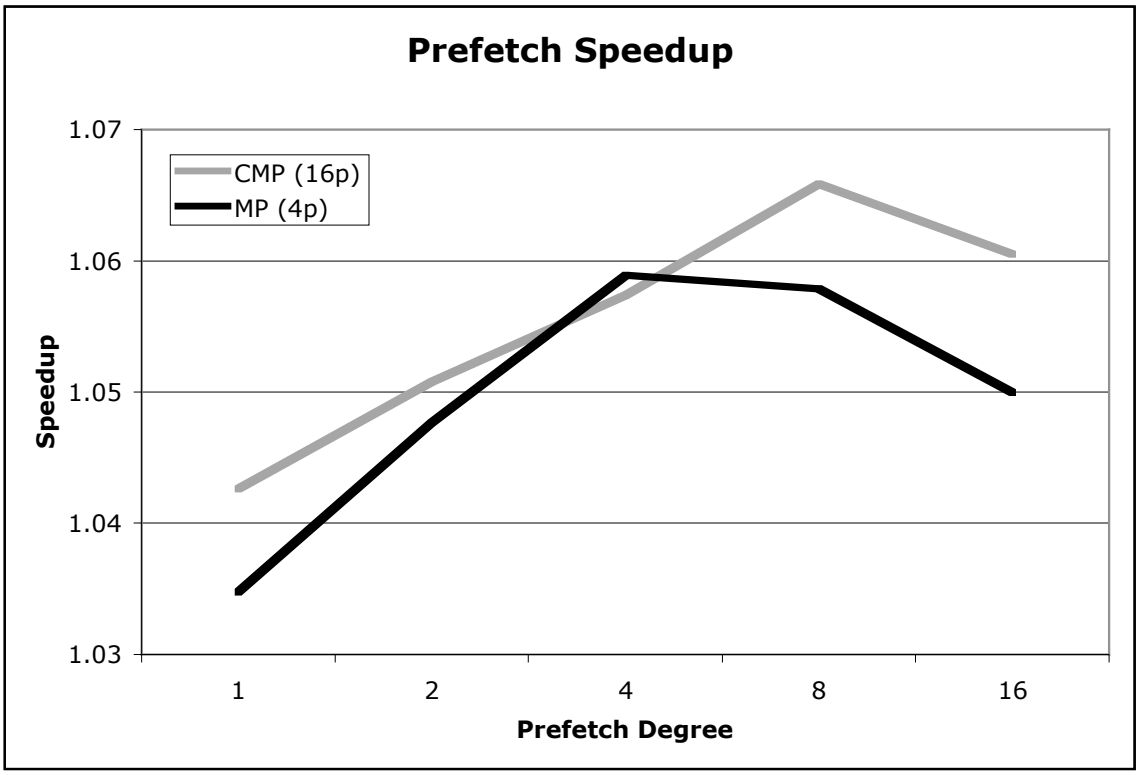


Figure 6: Prefetch Speedup

The final speedups are relatively good (at least when compared to our preliminary results). In fact, the CMP system performs better than the MP system. Initially, these results seem counterintuitive, but upon closer examination of simulation results, they are reasonable. Other simulation results show that link utilization in the CMP case is more consistent, that it has a smaller standard deviation. In contrast, the MP case has very erratic link utilization, a high standard deviation, and the links are often completely unused.

We would like to study this in more detail, but for now we assume that in the MP system, when a cluster of cache misses occurs many of the prefetches are dropped because the TBEs are half full. Then, once the memory requests return with data, there is a long period of time when the network links are completely unused, until another cluster of cache misses occurs. In the CMP system, clusters of cache misses occur more often, thus increasing link utilization, and reducing the standard deviation of the link utilization. In fact, the number prefetches (per unit work) that are issued to the interconnect network is more related to the number of TBEs than to the amount off-chip bandwidth. Overall, the more clusters of cache misses, the more prefetch requests that are sent to the interconnect network, and the better performance is. These results seem to favor CMP prefetching and our algorithm to drop prefetches based on TBE utilization.

5. Conclusion

In conclusion, we feel that CMP prefetching is very promising, more promising then uniprocessor prefetching. However, there are still many things that need to be examined in more detail, particularly prefetching GETX requests, since communication misses account for a large percentage of execution time. We also believe there are good opportunities from making the prefetching algorithm aware of network topology and the cache coherency protocol.

6. References

[1] A. Alameldeen, M.M.K. Martin, C. J. Maurer, M.Xu, M.D. Hill, D.A. Wood, D.J. Sorin, "Simulating a $2M Commercial Server on a $2K PC," IEEE Computer, Feb. 2003.

[2] J. F. Cantin and M. D. Hill. Cache, "Performance for Selected SPEC CPU2000 Benchmarks," Oct. 2001. http://www.cs.wisc.edu/multifacet/misc/spec2000cachedata/.

[3] M. J. Charney and T. R. Puzak, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *IBM Journal of Research and Development*, 31(3), 1997.

[4] T. Chen and J. Baer, "Effective hardware based data prefetching for high-performance processors," *IEEE Transactions on Computer Systems*, 44(5), pp. 609–623, May 1995.

[5] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "High-Performance DRAMS in Workstation Environments," *IEEE Transactions on Computer Systems,* 50(11), pp. 1133-1153, Nov. 2001.

[6] J. W. C. Fu and J. H. Patel, "Stride directed prefetching in scalar processors," In *Proceedings of the 25th Annual Symposium on Microarchitecture*, 1992.

[7] Z. Hu, M. Martonosi, S. Kaxiras, "TCP Tag Correlating Prefetchers," In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture, 2003.*

[8] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *IEEE Transactions on Computer Systems*, 48(2), pp. 121–133, 1999.

[9] N. Jouppi, "Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers," In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

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

[11] S. Kim and A. Veidenbaum, "Stride-directed Prefetching for Secondary Caches," *In Proceedings of the 1997 International Conference on Parallel Processing*, 1997.

[12] A. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," *In Proceedings of the 28th Annual International Symposium on Computer Architecture,* July 2001.

[13] K. Nesbit, and J. E. Smith, "Prefetching with a Global History Buffer," *In Proceedings of the 10th Annual International Symposium on High Performance Computer Architecture,* February 2004.

[14] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," *In Proceedings of the 21st Annual International Symposium on Computer Architecture,* May 1994.

[15] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceeding of the 30th Annual International Symposium on Computer Architecture*, Jun. 2003, pp. 336-347

[16] A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Transactions on Computers,* Vol. 11, No. 12, pp.7-21, Dec. 1978.

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

[18] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM Technical White Paper*, 2001.

[19] S. VanderWiel and D. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys,* 32(2), pp. 174–199, June 1999.

[20] Z. Wang, D. Burger, K. McKinley, S. Reinhardt, C. Weems, "Guided Region Prefetching: A Cooperative Hardware/Software Approach," *In Proceedings of the 30th Annual International Symposium on Computer Architecture,* June 2003.