Prefetching
6.911 Spring 2000

Ben Vandiver

May 10, 2000

1 Introduction

Prefetching is the art of hiding memory latency. Since processors are able to use data much faster than it can be retrieved, latency to memory can easily determine the speed of computation. Since the latency of retrieving a piece of data is generally proportional to the distance it is from the processor, an optimization would be to bring all the data you need closer to the processor. However, as access time drops, so does capacity, thus an algorithm must be implemented to keep only the required data close to processor. Numerous methods exist, and the key difference between them is whether the decision as to what data will be needed next is made in hardware or software.

The destination of prefetched values is an issue regardless of where prefetch decisions are made. Prefetched data is stored in one of three possible locations: registers, cache, or dedicated prefetch space. Placing prefetch result in a register, called a *binding prefetch*, is straightforward, but has problems with register allocation and data coherence. Since the register is designated when the prefetch is initiated, the register is bound up and useless until the prefetch completes. Secondly, a binding prefetch can cause a RAW hazard. Suppose the processor initiates a prefetch of location l which it needs at time t and completes the prefetch before time t and stores the result in a register. If it then writes location l also before time t, the value from the prefetch is now incorrect (stale). The second possible destination for a prefetch is the cache. This is called a *non-binding prefetch*, and although better than a binding prefetch, it still has problems with cache pollution. Cache pollution occurs when the prefetch engine forces useful data out of the cache with its prefetches. Lastly, data can be prefetched into dedicated structures, such as FIFO queues. Similarly to binding prefetches, dedicated structures have problems with data coherence. Additionally, these extra structures introduce more complexity into the processor, possibly lengthening the critical path.

2 Hardware-Driven Prefetch

In a hardware-driven scheme, hardware decides which memory addresses to prefetch based on past accesses or future instructions. Since the decision about what to prefetch is made at run time, it is less accurate than the compiler or programmer would be, thus leading to two types of problems: fetching the wrong data and fetching the data too late. Additionally, the extra logic required to make the prefetching decisions takes time to execute, which can lengthen the critical path.

The most common method of hardware-driven prefetch is caching itself. The cache prefetches data close to recent accesses, assuming that locality holds. If an application exhibits locality of reference, when it references location i, it increases the probability of subsequent access to locations in the neighborhood of i. In practice, many applications exhibit this property, partially because programmers and compilers generate code assuming a cache is going to be used.

Caches can fail spectacularly when applications do not exhibit locality or when applied in a multiprocessor environment. An example of an application which does not exhibit locality is one in which the working data set is larger than the cache size. In this case, the cache actually degrades performance because each data access will miss the cache, yielding no performance gain and incurring the penalty of checking the cache. In a multiprocessor, maintaining a consistent view of memory across all processors (cache coherency) becomes an issue. When one processor writes a location, that new value must be communicated to the other processors, which might have a (now stale) cache entry for that location.

Numerous methods have been developed to solve the cache coherency problem, where a solution is a protocol that maintains low latency access with minimal network overhead. One simple method, called Write-Invalidate, for cache coherency is to have the writing processor send an `invalidate` message to all other processors, which causes them to flush that location from their cache. This method fails to maintain low latency access when many processors are making many short writes to the same location (ex: semaphore), as each processor will need to re-fetch the location into the cache. Another method, called Write-Update, has the writing processor send the new value in an `update` message to each other processor. This method produces a lot of useless network overhead when processors are issuing long writes to differing locations. There exist a number of hybrid methods that store extra information on each cache line to help decide whether to issue an `invalidate` or `update` message [AK].

Other hardware prefetch methods examine the instruction stream to predict future accesses. Mostly, these engines attempt to optimize loops and vector operations because they have the best pain/performance payoff [Che95]. These methods attempt to predict upcoming accesses by knowing or guessing the stride. Since a loop is a sequence of operations repeated numerous times, memory accesses tend to be offset from each other by a constant amount, the stride. For example, adding a constant to a vector involves fetching each element of the vector. Therefore, the prefetch hardware issues loads for the locations it precomputes, and optimally executes the loop at a speed determined by memory bandwidth as opposed to memory latency. Since computing these guesses is difficult, the trend is to have the compiler give parameters to the prefetch hardware that give it hints about when and how loops are executed [cC94].

3 Software-Driven Prefetch

In software-driven prefetch the compiler issues explicit prefetch instructions. The prefetch instruction is a non-blocking, non-error-generating load. The advantage of software-driven prefetch is that the compiler can perform analysis on the code to determine where prefetching is needed and how far in advance to prefetch. The cost of this control is extra instructions, which must be fetched themselves, as well as clock cycles to execute those instructions.

However, the payoff can be worth the extra overhead, especially in distributed computing. The DASH system uses a prefetch strategy designed for a Non-Uniform-Memory-Access (NUMA) environment [MG91]. In the DASH system, each processor processor has its own cache. Processors are organized into groups, where each group is local to a block of memory and has a remote access cache and directory for managing remote data. Data is prefetched from remote memory and stored in a remote access cache local to each processor. This method is optimized to hide network latencies, and succeeded admirably in most cases. However, the authors discovered that improper prefetching can actually cause a performance hit due to "hotspotting." In this situation, all processors were prefetching the same block at the same time, which overloaded the owner of the block. The original DASH paper detailed results for hand-coded tests, however a later paper describes similar results using their new compiler [Mow98]. Additionally, the whole system is designed with cache coherency in mind to further reduce network overhead and access latency. Finally, a more relaxed consistency model allows for more reordering of loads to maximize the amount of prefetching.

4 Decoupled Access/Execute

Decoupled Access/Execute architectures take prefetching to its highest level, devoting a processor to nothing but accessing memory. In this architecture, a separate instruction stream is fed to the Access processor and the Execute processor, which execute asynchronously. The access processor performs all address calculations, loads, and stores. The execute processor receives data from the access processor, performs the given computation and sends the result back to the access processor for storage. The two processors communicate via FIFO queues, one pair for data exchange and the other to synchronize branches.

A DAE enjoys a number of advantages that enable it to perform very well. Since the Access processor tends to run ahead of the Execute processor, it hides the memory latency and allows the execute processor to run continuously, never stalling for data. In tests using hand-coded Lawrence Livermore loops run by Smith, the DAE architecture achieved an average speedup of 1.7 and a maximum speedup of 2.5 [Smi82]. A speedup of more than two for doubling the number processors is remarkable and it is achieved due the asynchronous execution of the two processors. If one processor stalls, the other processor continues executing, thus the processors do not spend time waiting to issue instructions. The only times both processors stall are for RAW hazards or full queues. By allowing an associative compare on the write address queue, the load queue in the Access processor only needs to be stalled for a matching address until the value for that address is produced by the Execute processor.

Additionally, a DAE architecture can outperform a three-way super scalar machine due to the nature of decoupled execution [FNN93]. The data exchange queues perform a type of register renaming. The slippage between the two asynchronous processors simulates out of order execution. Lastly, the slippage also performs dynamic loop unrolling because the Access processor leads the Execute processor, loading the values needed for a number of iterations of the loop before the Execute processor begins each loop body. Due to these factors, a DAE architecture outperformed a three-way super scalar machine on 10 out of 12 Lawrence Livermore loops.

More recently, decoupled architectures have been extended to include Control decoupling [BRT93]. When a dependency arises that forces the Access and Execute processors to synchronize, it is called a Loss of Decoupling (LOD) event. In an attempt to reduce LOD events, a new level of decoupling, control decoupling, was added that understands higher level branching, subroutine calls, and looping constructs. The Control processor is responsible for delivering instruction streams to the Access and Execute processors. In practice, control decoupling dramatically reduces the number of LOD events, each of which is compared to the cost of vector startup in the vector machine.

Finally, DAE architectures have a number of downside. First, the performance numbers quoted in the original DAE paper compare a DAE architecture with no caches to an ordinary machine with no caches [Smi82]. Introducing caches to the ordinary machine dramatically increases its performances. It also increases the performance of the DAE architecture, but not by as much [KnLC92]. Second, the dual (or triple) instruction streams are difficult to provide, and since they come from memory, they cause a performance hit. Lastly, programming a DAE architecture is non-intuitive and therefore requires a smart compiler. There are rumors of the existence of a DAE compilers, but nothing in the recent literature.

5 Conclusion

Memory latency will continue to be a problem in the foreseeable future. In the near future, the gap between processor speed and memory speed will continue to grow, making local latency hiding methods even more important. Currently, caches are the prefetching method of choice, but the power of more intelligent prefetching methods may draw interest. Intel has included a couple new features in its new Itanium architecture that give the compiler more power to fetch operands before they will be needed. On a wider scale, networked systems will always have to worry about prefetching simply because the speed of light makes a hard lower bound on the latency of remote access. By prefetching data and storing it physically closer to the machine, acceptable latencies can be acquired. Finally, prefetching involves communicating access patterns in some manner to the lower level architecture. As demonstrated by reconfigurable architectures, building flexibility into the hardware such that it can adapt to the type of computation being performed yields performance gains that cannot be ignored.

References

[AK] Craig Anderson and Anna Karlin. Two adaptive hypribd cache coherency protocols.

[BRT93] Peter Bird, Alasdair Rawsthorne, and Nigel P. Topham. The effectiveness of decoupling. In *Proceedings of the 1993 international conference on Supercomputing*, pages 47–56. ACM, 1993.

[cC94] Tzi cker Chiueh. Sunder: a programmable hardware prefetch architecture for numerical loops. In *Supercomputing Proceedings*, pages 488–497. IEEE, 1994.

[Che95] Tien-Fu Chen. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 237–242. IEEE, 1995.

[FNN93] Matthew Farrens, Pius Ng, and Phil Nico. A comparison of superscalar and decoupled access/execute architectures. In *Proceedings of the 28th Internationals Symposium on Microarchitecture*, pages 100–103. IEEE, 1993.

[KnLC92] Lizyamma Kurian and Paul Hulina nd Lee Coraor. Memory latency effects in decoupled architectures with a single memory module. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 236–245. ACM, 1992.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 1991.

[Mow98] Todd Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computing Systems*, 16(1):55–92, feb 1998.

[Smi82] James Smith. Decoupled access/execute computer architectures. *IEEE ISCA*, 1982.