Reducing the Impact of the Memory Wall for I/O Using Cache Injection

Edgar A. León, Kurt B. Ferreira, and Arthur B. Maccabe
Computer Science Department
The University of New Mexico
Albuquerque, NM 87131
{*leon,kurt,maccabe*}*@cs.unm.edu*

Abstract

Cache injection addresses the continuing disparity between processor and memory speeds by placing data into a processor's cache directly from the I/O bus. This disparity adversely affects the performance of memory bound applications including certain scientific computations, encryption, image processing, and some graphics applications. Cache injection can reduce memory latency and memory pressure for I/O. The performance of cache injection is dependent on several factors including timely usage of data, the amount of data, and the application's data usage patterns. We show that cache injection provides significant advantages over data prefetching by reducing the pressure on the memory controller by up to 96%. Despite its benefits, cache injection may degrade application performance due to early injection of data. To overcome this limitation, we propose injection policies to determine when and where to inject data. These policies are based on OS, compiler, and application information.

1. Introduction

Processor speeds continue to increase faster than memory speeds. This disparity adversely affects the performance of memory bound applications [21]. Computations with poor locality are typically limited by memory bandwidth. Examples of these computations include certain scientific applications, encryption, signal processing, string processing, image processing, and some graphics applications [14]. *Cache injection* [3, 9] is one of several techniques to alleviate the imbalance between processor and memory speeds [16, 17, 14]. This technique reduces memory latency and *memory pressure* (the number of requests issued to the memory controller per unit of time) by placing data from I/O devices directly into the cache as soon as data is available.

In current architectures, the transfer of data from I/O devices is done by writing to the system's main memory. When an application requests this data, the processor fetches it into the cache. *Prefetching* [16] anticipates accesses to blocks of memory based on usage patterns. With prefetching, we can hide memory latency by overlapping memory reads with computation. Unlike prefetching which uses memory bandwidth, cache injection reduces memory pressure by reducing the number of accesses to main memory.

The performance of cache injection is dependent on several factors including timely usage of data, the amount of data, and the application's data usage patterns. In a multiprocessor system, the consumer processor has to be identified so that data is written into the appropriate cache. If the application does not use the injected data promptly, cache injection may result in cache pollution, thereby evicting the application's working set from the cache. Therefore, the need for injection policies that determine the consumer processor and the target of data is crucial for optimal application performance.

In this paper we show that cache injection provides significant performance benefits over prefetching and without an appropriate policy can be harmful to application performance. We also propose injection policies based on OS, compiler, and application information that can overcome the limitations of cache injection. The rest of this paper is organized as follows. Cache injection is described in Section 2. An evaluation of cache injection and data prefetching is described in Section 3. In Section 4, we provide an example where cache injection without an appropriate policy decreases application performance. Section 5 describes injection policies based on OS, compiler and application information. Related work is described in Section 6. Finally, Section 7 states our conclusions and future work.

2. Cache injection

On current architectures, data from I/O devices is written to main memory. When this data is requested, the proces-

sor or prefetch engine fetches it into the cache. With cache injection, data from I/O devices is placed directly from the I/O bus into a processor's cache. This technique reduces memory latency by satisfying memory requests from cache, and it reduces memory pressure by reducing the number of requests to the memory controller.

Cache injection is a *producer-driven* and *non-binding* technique. It is producer-driven since the data transfer is initiated by the *producer* of data, in this case an I/O device. When a block of data is written or *injected* into the cache, it follows the cache's replacement and coherency protocol. This operation is called non-binding: data is not bound to a particular cache block in the cache.

Producer-driven mechanisms can be classified as implicit or explicit [5] depending whether the producer knows the identity of the consumer. Cache injection is an *explicit* method, the consumer or target of data has to be identified before the injection operation takes place. The target can be a level 2 cache, level 3 cache, or main memory. Also, the consumer processor has to be identified to determine the appropriate cache or memory. In the remainder of this paper, we use cache injection of incoming network messages to provide a specific example of this technique, even though cache injection can be used with other DMA devices.

In Figure 1, a network interface controller (NIC) writes incoming network data to main memory. When this data is needed, it is fetched by the processor incurring memory latency due to the impact of the memory wall. Although this latency can be overlapped with computation by using prefetching, memory bandwidth is still consumed.

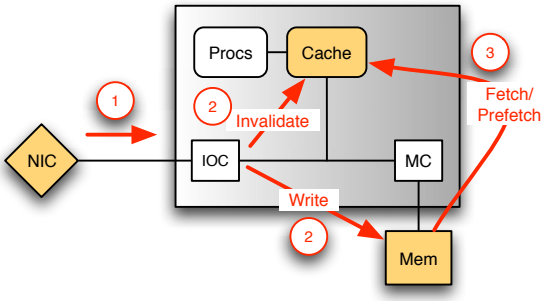

Figure 1. Memory write operation initiated by the NIC. In step 1, incoming network data arrives at the NIC which in turn initiates the transfer to memory through the IO controller (IOC); in step 2 cached copies are invalidated and data is written to main memory through the memory controller (MC); in step 3, the processor or the prefetch engine fetches data from main memory into the cache.

As Figure 2 shows, cache injection transfers incoming network data directly from the NIC to a processor's cache. When this data is used promptly by the processor, memory latency and memory pressure are significantly reduced. Fetching incoming network data from memory is no longer necessary and thus requests issued to the memory controller are reduced.

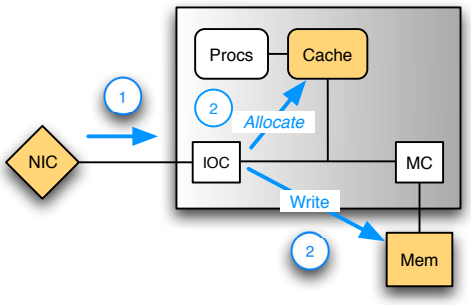

Figure 2. Cache injection operation initiated by the NIC. Unlike the memory write operation, step 2 allocates incoming network data into the cache. If the processor uses this data promptly, there is no need to fetch it from main memory.

3. Comparing cache injection and prefetching

Cache injection and data prefetching strive to reduce memory latency by moving data into the cache before it is needed. Unlike prefetching which provides a general technique to reduce memory latency, cache injection can only be applied to data from I/O devices. Prefetching is a consumer-driven technique while cache injection is producer-driven.

Many studies have shown that prefetching is an effective technique to reduce memory latency. However, it has several disadvantages when compared to cache injection. First, prefetching data from I/O devices incurs memory bandwidth due to two transactions: (1) transfer of data from the I/O producer to memory, and invalidating cached copies; and (2) fetching data from memory to the consumer. With cache injection, the second transaction is not necessary. Second, even if prefetching correctly anticipates the access to I/O data, fetch requests can only be served from main memory, incurring memory latency as well as using memory bandwidth.

It is important to note that both cache injection and prefetching may not perform optimally. With prefetching, data may be fetched too late to reduce memory latency. With cache injection, data may be injected into the cache

too early. This early injection of data may evict part of the application's working set. The working set has to be brought back to cache from lower levels of the memory hierarchy incurring additional overhead. This overhead is studied in Section 4. Table 1 summarizes the differences of prefetching and cache injection. The next two subsections describe the experimental framework, methodology, and results of our evaluation of these two techniques.

Table 1. Prefetching vs. cache injection.

	Prefetching	Cache injection
Resources	1) write to memory 2) fetch to cache	1) write to cache
	incurs memory latency and bw usage	reduces memory latency and bw usage
Fails when	fetch too late	inject too early
Applicability	general-purpose	limited to I/O
Comm.	consumer-driven	producer-driven

3.1. Experimental framework

The experimental infrastructure is based on simulation and consists of two components: the base architecture and a high-performance communication system. The base architecture is based on IBM's *Mambo* full-system simulator [2]. Mambo has been extended with an implementation of cache injection to the L3 cache [3]. The simulated machine is a multi-core, cache-coherent, distributed shared memory architecture.

The high-performance communication system consists of a simulated high-performance NIC and an OS-bypass zero-copy network stack [13]. The NIC is capable of running arbitrary functionality. It interacts with the host system through conventional memory write operations and through non-binding cache injection operations. The NIC is also capable of moving data into any level 2, level 3 caches or main memory.

The network stack provides an unreliable datagram connectionless service with a UDP-like interface. We implemented this interface using an OS-bypass, zero-copy design which is common in high-performance networks. Our implementation, *Fast UDP*, consists of code running on the host and the NIC. The code running on the host is a user-level library that virtualizes NIC resources to the application. The code running on the NIC implements message matching and checksum processing.

3.2. Experimental evaluation

We measure memory bandwidth and execution time of an application in three configurations: (1) base case with no optimizations; (2) base configuration with prefetching; and (3) base configuration with cache injection [12]. The application used in this evaluation performs a linear traversal of incoming network data in calculating a reduction operation. We chose this application because: (1) it represents a stage of computation that is limited by memory bandwidth; and (2) it provides an optimal case for prefetching (linear traversal of data).

The machine configuration for these experiments is shown in Table 2. The simulated machine is a Power5 architecture [19] with a non-binding cache injection implementation to the level 3 cache. The Power5 processor chip includes two cores, a level 1 cache per core, a shared level 2 cache, a memory controller, and an I/O controller. The level 3 cache is a *victim cache* [8] and is implemented off-chip. Data prefetching is implemented in hardware by the architecture. Data is prefetched into the L1 data cache by first fetching it into the L2 cache and then from the L2 to the L1 cache.

Table 2. System configuration parameters.

Simulator	Mambo PowerPC full-system simulator
OS	K42 w/OS-bypass zero-copy Fast UDP
Architecture	Power5 with cache injection to L3
Processor	1.65GHz frequency
L1 I/D cache	64KB/32KB 2-way/4-way
L2 cache	1.875MB 3-slice 10-way 10 cycle latency
L3 cache	36MB 3-slice 12-way 80 cycle latency
Cache line	128B
Main memory	512MB 230 cycle latency

First, we measure the memory bandwidth used by the application in terms of the number of memory reads requested to the memory controller. As shown in Figure 3, the base case and the prefetching configuration perform equally as prefetching has to fetch incoming network data from memory. Prefetching anticipates data accesses correctly due to the sequential access pattern used by the application. Cache injection significantly reduces the number of memory reads by up to 96% as all application accesses to incoming network data hit the L3 cache.

Second, we measure the execution time of the application in processor cycles. As shown in Figure 4, cache injection and prefetching outperform the base case as they both reduce the number of cache misses. Prefetching reduces execution time by up to 37% while cache injection by up to 30%. Prefetching performs better because it fetches blocks to the L2 cache, while our cache injection implementation targets the L3 cache. We are investigating the impact of injecting data into the L2 cache. Given its lower latency, we expect that injections to the L2 will perform as well as prefetching.

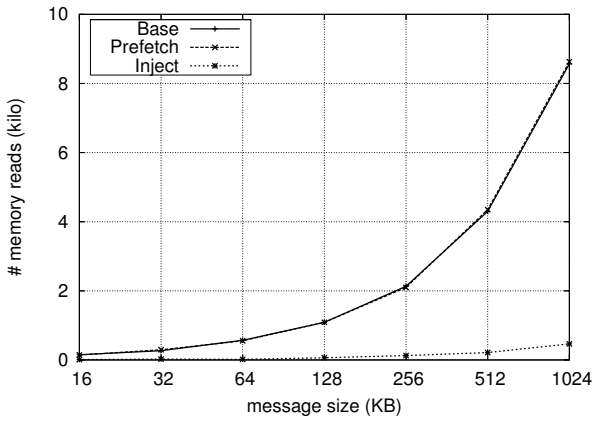

Figure 3. Memory bandwidth utilization.

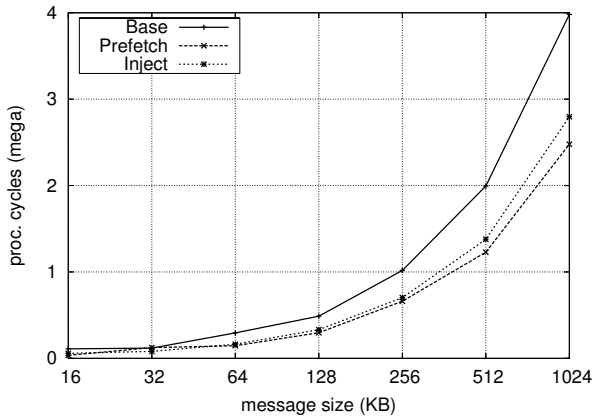

Figure 4. Execution time.

Since cache injection reduces memory pressure for incoming network data, its benefits are dependent on the ratio of incoming network data and local data used by the application. The communication traffic and granularity of communication vary from application to application and thus the improvements on performance will vary. Several high-performance computing applications will likely benefit from cache injection as they exchange a significant amount of network messages. For example, SMG2000 [4], a memory intensive application, at 384 tasks spends almost 75% of the overall application aggregate time in communication operations [20].

4. Blind cache injection

In the previous section, we showed that cache injection reduces memory pressure significantly. Also, previous work [3, 9], has shown that cache injection can provide significant performance improvements for a particular type of application, namely TCP/IP protocol processing. In this case, the kernel consumes the injected data right after it is written, signaled by an interrupt to the processor.

Cache injection, however, presents challenges intrinsic to the explicit producer-driven nature of this technique, namely timely transfer and identifying the consumer of data. With cache injection, data may be transfered too early for the consumer to process it. We refer to this type of data movement as *blind transfer* since data is transfered without knowledge of the usage patterns of the consumer. *Blind cache injection* may create cache pollution taking useful data out of the cache.

Cache injection requires explicit knowledge about the identity of the consumer. The NIC, the producer of data, has to choose between a set of potential consumers. Even in a uniprocessor system, the choice between a level 2 and level 3 cache has to be made. In a multiprocessor system, a chip, core or processor has to be chosen. An incorrect choice of the target may result in higher delays than just moving data directly into main memory.

In the remaining of this section, we show an application in which cache injection without an appropriate injection policy (i.e., blind injection) is harmful to performance. This motivates the study of injection policies, which are directly related to the effectiveness of this technique.

4.1. The Jacobi method

The *Jacobi method* [7] is an iterative algorithm to solve a partial differential equation called the *Laplace equation*. An example application of this algorithm is the temperature calculation of a body represented by a multidimensional grid. At each time step, the Jacobi method computes the temperature of all interior points based on their neighbors temperature. The algorithm continues to refine the temperature values until a specific threshold is reached. The *edge* or *boundary* points are fixed and set by boundary conditions.

In a two-dimensional grid (n, n), we can partition the problem domain into sub-domains that can be assigned to individual processes. This domain decomposition provides the basis for a parallel implementation. Given p processes, the grid is decomposed into sets of n/p rows. Every process is in charge of computing n^2/p points. To compute the values in the first and last row for a particular process, the values of the boundary rows from its previous and next neighbor processes are needed. This requires data exchange between processes: at every time step each process sends the values of its first and last row ($2N$ values) to its neighboring processes accordingly. In a message-passing communication paradigm such as MPI [6], an algorithm (per process per iteration) that overlaps computation and communication can be outlined as follows:

1. `MPI_Isend` boundary data

2. `MPI_Irecv` boundary data

3. Perform local computation

4. `MPI_Wait` for remote data to arrive

5. Perform remote data computation

4.2. Performance using blind injection

The steps of the algorithm outlined above can be classified into *communication*, *computation* and *delay* stages. Steps 1 and 2 are communication steps, steps 3 and 5 are computation steps, and step 4 is a delay step.

Assume that the algorithm is running in a cluster of nodes, where one process runs in one node and every node is connected through a NIC to the cluster's local network. At the beginning of execution, every process executes the communication steps. Since the communication operations are non-blocking, every process continues executing step 3 even if the previous operations have not been completed. Also, assume that while executing step 3, data arrives to the NIC and is written to main memory (overlap of computation and communication). If step 3 takes long enough for the communication stage to complete, then step 4 completes immediately and we continue to execute the last step. Otherwise, the process waits for the requested data to arrive at which point moves to the last step.

In a system with blind cache injection, i.e., data is moved directly from the NIC to the processor's cache as soon as it arrives, the overlap of communication and computation steps may actually be problematic. While the processor is working on local data, the NIC cache injection operation may be taking the application's working set out of the cache. Thus, the application has to fetch again its working set possibly replacing those blocks written by the NIC displacing network data back to main memory. After completion of step 3 and if all the network data has arrived, the processor has to fetch the network data back to the cache to execute the last step. The effects of blind injection in this case result in the overhead of unnecessarily evicting the application's working set from the cache, and then fetching this data back to the cache. This overhead increases memory bandwidth traffic as well as memory latency. The conventional system without cache injection performs better.

The performance penalty incurred by blind injection is due to the producer-driven nature of this technique. In other words, data is written to the cache as soon as it is produced (when it arrives from the network), which happens to be too early for the application to take advantage of it. To leverage the memory latency and memory bandwidth benefits of this technique we need injection policies that allow the NIC to make smarter decisions about when to inject into the cache.

Two simple policies can overcome this problem: (1) for incoming network data which size exceeds a specific threshold, write data to the L3 cache, otherwise write data to the L2 cache; and (2) if the consumer thread or process is blocked waiting for data, write into the L2, otherwise write to main memory. The first policy is appealing considering the growing size of level 3 caches, for example a Power5 machine contains a 36MB L3 cache. The second policy which requires OS information, performs as well as the conventional system with the added benefit that if the application is blocked waiting for data, it will perform better by avoiding the added overhead of memory latency.

Thus, if an application does not use the injected data promptly, blind cache injection may create cache pollution resulting in loss of performance. The performance benefits of this technique rely on a *good injection policy*. This policy is dependent upon the usage pattern of an application, the OS and the compiler.

5. Injection policies

To leverage the performance improvements that can be provided by cache injection, adequate policies are needed. The goal of these policies is to determine the target device where the data will be consumed. This target can be either a level 2 cache, a level 3 cache, or main memory. In a multiprocessor system, a policy also determines which processor will consume the data. We have identified a set of policies which we believe will enhance the performance of applications using cache injection. The following policies are based on information from the OS, the compiler, and the application.

Processor-direction. Inject to the processor/cache indicated by the memory descriptor posted by the application. The memory descriptors residing on the NIC to match incoming messages have information about the consumer of data. When a buffer is registered by the application with the kernel, the OS adds the processor identifier where the application's thread is running to this buffer's memory descriptor. If threads migrate in the system, the OS can update the NIC with this information so that messages can be routed to the appropriate processor.

Size-dependent. Inject to the level 2 cache, level 3 cache, or main memory depending on the size of the incoming message. Inject to the L2 cache if message size is below a *cache threshold*, to the L3 cache if size is between the cache threshold and a *memory threshold*, and to main memory if size is over the memory threshold.

In-cache. Inject to the target cache if the line is in it. Searches in a cache can be an expensive operation since the whole cache may be traversed. *Smart searches* [10] can be employed to determine rapidly if a line is not in the cache. However, the search algorithm is prone to false-hits. In

other words, the search may indicate a hit when the line is not cached. Smart searches use an array located in the cache controller to store partial tag bits. Depending on the number of bits, false-hits are less probable to occur at the cost of extra space in the smart search array.

On-wait. Inject to the target cache when the consumer thread is blocked waiting for incoming network data. The MPI implementation can notify the NIC when the application is about to block waiting for network data. To do so, the message descriptor identifier or token is also passed to the NIC. When the message arrives from the network, the NIC writes to the cache if the memory descriptor for that message indicates that the application is waiting. Otherwise, data is written to main memory.

Compiler-driven. Inject to the target cache when the application and/or compiler explicitly solicits the data. This policy uses *software injection* which is analogous to software prefetching. With software injection, hints can be passed to the NIC to indicate that specific messages should be injected into cache. These hints can be automatically generated from the compiler using software prefetching techniques, or they can be specifically indicated by the application.

6. Related work

Several techniques currently exist to manage the imbalance between processor and memory speeds. Data caching reduces memory latency for data access patterns with good locality. With prefetching, memory latency can be overlapped with computation to improve processor performance. However, prefetching exhibits the same shortcomings of caching for poor locality computations. Furthermore, it does not improve memory bandwidth. Software access ordering [17] improves memory bandwidth by changing the order of memory accesses at compile time. This technique, however, is limited to static information and cannot take advantage of run-time access patterns of data. Hardware-assisted access ordering [14] decouples the order of requests issued by the processor from those issued to the memory system. This technique strives at minimizing the average latency over a coherent set of accesses dynamically.

Unlike cache injection, all of these techniques incur memory latency and memory bandwidth usage for data from I/O devices. Data has to be fetched from main memory when requested by a processor. Cache injection can reduce memory latency and memory pressure by placing data directly into the cache from the I/O bus. Cache injection can be effective even for computations with poor locality.

Bohrer [3] proposed and analyzed cache injection as a mechanism for reducing memory latency. The authors used this technique on TCP/IP protocol processing. Using micro-benchmarks for network and disk communication they showed significant improvements on execution time. The work by Huggahalli [9] improves upon Bohrer's work by showing a significant reduction in both memory latency and memory bandwidth usage for benchmarks such as SPECWeb9, TPC-W and TPC-C.

In these two studies, data is injected into the cache blindly and it does not adapt to the application's needs. In some instances, writing to cache may not be desirable. The processing of TCP/IP packets is a desirable target for blind injection since the kernel consumes the injected packet right after it is written (signaled by an interrupt to the processor). However, as we have discussed, blind injection is prone to degradation of application performance.

Cache injection is one of several producer-driven nonbinding techniques to reduce memory latency between processors. The Standford Dash multiprocessor [11] included two producer-initiated operations: `update-write` and `deliver`. The former writes data directly to all processors' caches that have cached the data, while the latter to a specific group of processors. In Poulsen's work [18], data is forwarded between processors to optimize shared accesses of data. Data forwarding was implemented using the `forwarding write` operation. Abdel-Shafi's work [1] also uses data forwarding in the form of `remote write` operations to reduce memory latency for fine-grain communication.

Milenkovic [15] uses a combination of data forwarding and prefetching. The consumer uses a prefetch-like instruction (`lprefetch`) to fetch data likely to be used in the near future. Unlike prefetching, this instruction only records the requested data in an *injection list* that resides in the cache controller. At the producer side, a forward-like instruction (`write_back`) is used to send data over the bus when it becomes available. Consumers snoop the bus and store the data if it matches an entry in the injection list. To my knowledge, the work by Milenkovic on bus based multiprocessors is the first to use the term cache injection.

The difference between data forwarding and our work on cache injection is the producer and consumer of data. In data forwarding both processors reside in the same computational node. In cache injection the producer and consumer processors reside in different computational nodes linked together by a high performance network. This architectural difference must be accounted for in the timings of injection policies. Some ideas used in data forwarding about when to inject into the cache can also be applied to cache injection.

7. Conclusions and future work

In this paper we have shown how cache injection outperforms prefetching in reducing the impact of the memory wall on applications. More specifically, cache injection reduces memory latency and memory pressure by placing

data from I/O devices directly into the cache. This reduction in memory pressure stems from a reduction in the number of requests issued to the memory controller. The performance of cache injection is dependent on several factors including timely usage of data, the amount of data, and the application's data usage patterns. Without an appropriate policy to determine when to inject into the cache, cache injection may degrade application performance. We believe that these policies will be based on information from the OS, the compiler, and the application. We have proposed certain policies based on this information which we plan to implement and evaluate. We are also characterizing those scenarios where cache injection is appropriate, and characterizing the types of applications that will benefit from this promising optimization technique.

Acknowledgments

We would like to thank Orran Krieger, Michal Ostrowski, Lixin Zhang, and James Peterson from IBM Research; Hazim Shafi from Microsoft Research; and Patrick Widener from the University of New Mexico. This work has been supported by an Intel Fellowship and an IBM grant.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97)*, pages 204–215, San Antonio, TX, 1997.

[2] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo – a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, Mar. 2004.

[3] P. Bohrer, R. Rajamony, and H. Shafi. Method and apparatus for accelerating Input/Output processing using cache injections, Mar. 2004. US Patent No. US 6,711,650 B1.

[4] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[5] G. T. Byrd and M. J. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3):456–466, 1999.

[6] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Knoxville, TN, 1994.

[7] F. R. Gantmacher. *The Theory of Matrices*, volume I. AMS Chelsea, 1959. Translated from Russian.

[8] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. D. Maciel, C. Sosa, and R. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbooks, second edition, 2005.

[9] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, Madison, WI, June 2005.

[10] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 211–222, San Jose, CA, Oct. 2002.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[12] E. A. León and A. B. Maccabe. Reducing memory bandwidth for chip-multiprocessors using cache injection. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06). Poster Session*, Seattle, WA, Nov. 2006.

[13] E. A. León and M. Ostrowski. An infrastructure for the development of kernel network services. In *20th ACM Symposium on Operating Systems Principles (SOSP'05). Poster Session*, Brighton, United Kingdom, Oct. 2005. ACM SIGOPS.

[14] S. A. McKee, S. A. Moyer, and W. A. Wulf. Increasing memory bandwidth for vector computations. In *International Conference on Programming Languages and System Architectures*, pages 87–104, Zurich, Switzerland, Mar. 1994.

[15] A. Milenkovic and V. Milutinovic. Cache injection on bus based multiprocessors. In *17th Symposium on Reliable Distributed Systems (SRDS'98)*, pages 341–346, West Lafayette, IN, 1998.

[16] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[17] S. A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, Department of Computer Science, University of Virginia, Apr. 1993.

[18] D. K. Poulsen and P.-C. Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *International Conference on Parallel Processing (ICPP'94)*, pages 276–280, North Carolina State University, NC, 1994.

[19] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[20] J. S. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. In *2002 ACM/IEEE Conference on Supercomputing (SC'02)*, pages 1–18, Baltimore, Maryland, 2002.

[21] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 3(1):20–24, Mar. 1995.