5-1-2009

Improving the performance of parallel scientific applications using cache injection

Edgar Leon Borja

Edgar A. Leon Borja

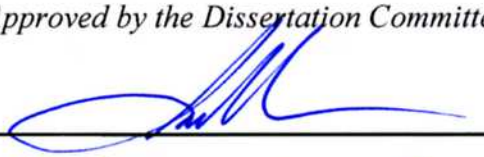
Candidate

Computer Science

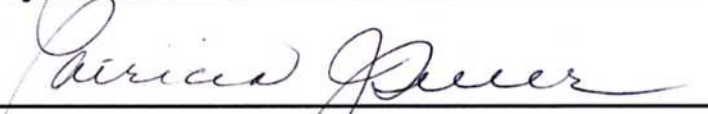
Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:



, Chairperson



Improving the Performance of Parallel Scientific Applications Using Cache Injection

by

Edgar A. León Borja

B.S., Computer Science,
Universidad Nacional Autónoma de México, 2001

M.S., Computer Science, University of New Mexico, 2003

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2009

Dedication

To my parents, for their endless efforts in looking out for my happiness.

Acknowledgments

Foremost, I would like to thank my adviser, Barney Maccabe, for his mentoring, advise, guidance, encouragement, friendship, and for his key role in my development as a scientist. I would also like to thank Rolf Riesen for his contributions to the scalable cluster simulator, for setting it up at Sandia, and for executing many experiments there; Ron Brightwell for allowing me to use his MIAMI API and his MPICH device layer; Kurt Ferreira for his feedback during our many technical discussions and for his friendship; Michal Ostrowski for his contributions to the shim layer; Orran Krieger and Hazim Shafi for their feedback and for introducing me to cache injection; Lixin Zhang and Jim Peterson for their support in using Mambo; Amos Waterland and Jimi Xenidis for their feedback, support and friendship; Dilma da Silva for being part of my dissertation committee and for her feedback and friendship; Patricia Teller for being part of my dissertation committee and for her feedback; Linda Maccabe for her friendship and moral support in my education; the Brantley family and the Calderon family for their love and friendship; Katie Edwards for her editorial comments to this document; the members of the Scalable Systems Laboratory for their support; and finally, the Department of Energy, Office of Science, IBM and Intel for their generous support in this research.

Improving the Performance of Parallel Scientific Applications Using Cache Injection

by

Edgar A. León Borja

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2009

Improving the Performance of Parallel Scientific Applications Using Cache Injection

by

Edgar A. León Borja

B.S., Computer Science,

Universidad Nacional Autónoma de México, 2001

M.S., Computer Science, University of New Mexico, 2003

Ph.D., Computer Science, University of New Mexico, 2009

Abstract

Cache injection is a viable technique to improve the performance of data-intensive parallel applications. This dissertation characterizes cache injection of incoming network data in terms of parallel application performance. My results show that the benefit of this technique is dependent on: the ratio of processor speed to memory speed, the cache injection policy, and the application's communication characteristics.

Cache injection addresses the memory wall for I/O by writing data into a processor's cache directly from the I/O bus. This technique, unlike data prefetching, reduces the number of reads served by the memory unit. This reduction is significant for data-intensive applications whose performance is dominated by compulsory cache misses and cannot be alleviated by traditional caching systems.

Unlike previous work on cache injection which focused on reducing host network stack overhead incurred by memory copies, I show that applications can directly benefit from this technique based on their temporal and spatial locality in accessing incoming network data. I also show that the performance of cache injection is directly proportional to the ratio of processor speed to memory speed. In other words, systems with a memory wall can provide significantly better performance with cache injection and an appropriate injection policy. This result implies that multi-core and many-core architectures would benefit from this technique. Finally, my results show that the application's communication characteristics are key to cache injection performance. For example, cache injection can improve the performance of certain collective communication operations by up to 20% as a function of message size.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For almost two decades, the growing disparity of processor to memory speed has affected applications with poor temporal locality in their ability to benefit from improvements in processor speed. This disparity known as the memory wall [58], makes memory speed the limiting factor in the performance of a system. Even in systems with perfect caches, the memory wall affects application performance due to compulsory cache misses on data with poor locality. Data prefetching may not alleviate this problem in applications where the memory bus is already saturated, or those with not enough computation in between memory accesses to mask memory latency.

Examples of applications affected by the memory wall include scientific, cryptographic, signal processing, string processing, image processing, and some graphics computations [42]. More recently, many high-end, real-world scientific applications showed a strong dependence on the memory characteristics of a system [48, 47]. These applications show poor locality by accessing large amounts of unique data (data-intensive applications). This data generates compulsory cache misses resulting in an increased dependency on memory speed.

Cache injection alleviates the memory wall by placing data from an I/O device directly

into the cache [6, 28, 12]. The benefits of this technique are reducing memory access latency and reducing memory pressure (the number of requests issued to the memory controller per unit of time). In current architectures, I/O data is transferred to main memory, and cached copies of old values are invalidated. Accessing I/O data results in compulsory cache misses and, thereby, accesses to main memory. To hide memory latency, prefetching can overlap memory accesses with computation [4, 44]. Prefetching anticipates memory accesses based on usage patterns or specific instructions issued by the compiler, the OS or the application. Prefetching can hide memory latency, but unlike cache injection, it does not reduce and may increase traffic over the already saturated memory bus, a precious resource for memory-bound applications. Prefetching is more widely applicable than cache injection, but the latter provides better performance for I/O. Recent Intel architectures provide a similar mechanism to cache injection called Prefetch Hint [51]. This mechanism allows early prefetching of I/O data initiated by the NIC. Like prefetching, this technique does not reduce memory bandwidth utilization.

The impact of cache injection on application performance is dependent on several factors including timely use of data, the amount of data, the type and frequency of communication primitives, the application's data usage patterns, and the underlying architecture. For example, injecting data into a cache may evict the application's working set, and in a multiprocessor system, an incorrect choice of processor/cache may increase overhead (local memory may be closer than another processor's cache).

Appropriate injection policies must be made to account for these factors. Injection policies answer the questions of *what*, *when*, and *where* to inject. The impact of cache injection on application performance is dependent on the policies that determine the consumer processor/core and the appropriate level in the memory hierarchy. The information needed by these policies is distributed throughout the system, in the application, the OS, the communication library, the compiler and the caches. For example, the mapping of processes to processors/cores is held by the OS; the application and/or the compiler can

provide hints of data usage; information about the implementation of collective communication operations may reside in the communication library and/or the NIC.

In this work, I show how cache injection can improve the performance of parallel applications as a function of processor to memory speed ratio, injection policy, and communication characteristics of applications. This characterization of cache injection provides a framework to identify applications which may benefit from this technique.

In the next section, I present my thesis statement and contributions of this work, followed by a description of my research approach. An outline of the overall thesis is presented in Section 1.3.

1.1 Thesis statement and contributions of this work

> *Cache injection can improve the performance of parallel scientific applications as a function of the ratio of processor to memory speed, the injection policy, and the application's communication characteristics.*

The major contribution of this work is to show that cache injection can improve the performance of parallel scientific applications. This improvement is dependent on: (1) the processor to memory speed ratio; (2) the injection policy; and (3) the communication characteristics of applications. Cache injection addresses the memory wall for data intensive applications with a significant amount of communication.

To adequately build the foundations of this work, I developed micro-benchmarks to investigate the benefits and limitations of cache injection and created a scalable infrastructure to analyze the impact of this technique on application performance. I compared and contrasted the benefits of cache injection and data prefetching. I showed that, unlike data prefetching, cache injection can reduce memory bandwidth utilization. To illustrate the

main limitation of cache injection, I showed an example where this technique may create cache pollution. To address this limitation, I proposed cache injection policies based on OS, application and compiler information. I implemented a subset of these policies tailored for MPI and analyzed their impact on application performance using an infrastructure to study the impact of cache injection and other novel architectural features on application performance at scale.

A more detailed list of the contributions of this work follow:

- The performance of cache injection is directly proportional to the processor to memory speed ratio. The higher the memory wall, the greater the benefit.
- Injecting communication meta-data improves the performance of latency-sensitive applications.
- Injecting application data improves the performance of bandwidth-sensitive applications that show temporal and spatial locality in using incoming network data.
- Cache injection can improve the performance of applications using a significant number of collective operations. Cache injection can improve the performance of MPI_Allgather, MPI_Alltoall, MPI_Allreduce, and MPI_Bcast operations.

This work is unique in relating cache injection of application data to application performance. Previous work focused on using cache injection to reduce the overhead of memory copies incurred by the network stack. High-performance communication systems typically use zero-copy implementations which do not benefit from this.

1.2 Research approach

Using simulation, I characterize the impact of several injection policies on the performance of two applications. The following steps summarize my research approach.

1. Characterize the benefits of cache injection. I developed a micro-benchmark to show, experimentally, the benefits of cache injection compared to data prefetching. These experiments measured execution time and memory pressure incurred by the micro-benchmark. Unlike data prefetching, cache injection reduces memory bandwidth utilization [13, 14].

2. Characterize the limitations of cache injection. Using the Jacobi iteration, I demonstrated, analytically, that cache injection can pollute the cache and decrease application performance [12]. In this case, the NIC and the application compete for the cache. The working set of the application may be evicted by incoming network data written by the NIC.

3. Design injection policies to address the limitations of cache injection. I proposed a set of policies to minimize the amount of pollution introduced into the cache. These policies answer the questions of what, when, and where to inject. Using simulation, I implemented a subset of policies tailored for MPI.

4. Design and implement a scalable infrastructure to execute parallel scientific applications on a cluster of cache injection nodes. I coupled hundreds of instances of an existing cycle-accurate, full-system simulator with cache injection using a model of a high-performance network [18]. I showed that the resulting distributed infrastructure can accurately simulate application performance on a parallel, high-performance machine [19].

5. Characterize the impact of cache injection on parallel application performance. Using the cluster simulator mentioned above, I measured the performance of selected applications using a set of injection policies tailored for MPI. The results showed that cache injection is particularly effective on machines with a memory wall. Furthermore, the performance of cache injection is dependent on the application's communication characteristics and the injection policy.

1.3 Thesis outline

The rest of this document is organized as follows. Chapter 2 presents a detailed study of cache injection. This study includes a demonstration of an upper bound on the performance benefits of this technique; a comparison with data prefetching; an example where cache injection without an appropriate policy can be decremental to application performance; and a description of policies based on OS, compiler, communication library, and application information to improve the performance of applications. Chapter 3 describes the experimental infrastructure developed to execute parallel applications using MPI with and without cache injection. This infrastructure is based on simulation and leverages current cycle-accurate simulators into a distributed and scalable cluster simulator. Even though the simulated system cannot be validated against a real machine (because it does not exist), I developed a set of experiments using unmodified parallel applications that show the cluster simulator is accurate. Chapter 4 describes the test environment including the platform used for running the cluster simulator, the simulated system configuration, the injection policies implemented, and the applications and analysis tools used in this study. Chapter 5 illustrates the impact of different injection policies on application performance. It also analyzes the relationship between cache injection performance and the ratio of processor to memory speed and the application's communication characteristics. Chapter 6 describes related work. Finally, Chapter 7 summarizes the results and contributions of this work, as well as a discussion of possible directions for future research.

Chapter 2

Cache injection

Cache injection [6, 28] is one of several techniques to mitigate the imbalance between processor and memory speeds [44, 46, 42]. This technique reduces memory access latency and memory pressure by placing data from I/O devices directly into the cache.

In current architectures, data from I/O devices is written to the system's main memory. When an application requests this data, the processor fetches it into a local cache. Fetching data can be done ahead of time by a prefetch engine which may anticipate accesses to blocks of memory based on usage patterns. With prefetching [44], data latency is reduced by overlapping memory reads with computation. Unlike prefetching, 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, evicting the application's working set from the cache. This motivates the need for injection policies that determine the consumer processor and the appropriate level of the memory hierarchy where data may be written (L2

cache, L3 cache or main memory). The information needed for such policies may come from different sources including the OS, the compiler, the application, and the communication library. For example, the OS is responsible for assigning software threads/processes to cores/processors and maintaining this information.

In this chapter, I provide: (1) experimental results showing that cache injection can reduce memory bandwidth utilization as compared to data prefetching; (2) an example where cache injection without an appropriate policy can be harmful to application performance; and (3) injection policies based on OS, compiler, and application information that can overcome the limitations of this technique.

2.1 What is cache injection?

In current architectures, data from I/O devices is written to main memory. Before the processor can use this data, it is fetched to the cache by the processor or by a prefetch engine. With cache injection, data from I/O devices is placed directly from the I/O bus into a processor's cache. Cache injection 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 because 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 non-binding because data is not bound to a particular block in the cache.

Producer-driven mechanisms can be classified as implicit or explicit [10] depending on whether the producer knows the identity of the consumer. Cache injection is an explicit method, the consumer or target of data must be identified before the injection operation

takes place. The target can be an L2 cache, L3 cache, or main memory. Also, the consumer processor must be identified to determine the appropriate cache or memory. In the remainder of this document, I 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.

Figure 2.1 depicts the steps a traditional cache-coherent architecture follows when receiving data from the network. For each cache block, a write-invalidate operation is performed, i.e., write to memory and invalidate the appropriate cache block. This operation results in compulsory cache misses, thereby incurring memory latency for incoming network data. Although prefetching can overlap memory latency with computation, memory bandwidth is still used.

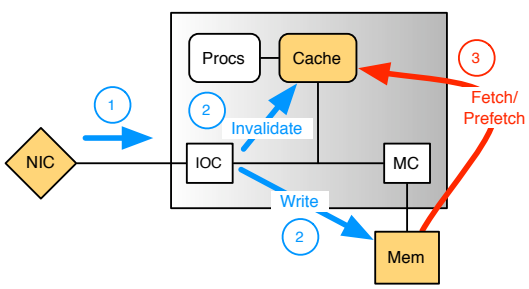


Figure 2.1: Memory write operation initiated by the NIC. In step 1, incoming network data arrives at the NIC which in turn initiates the transfer to memory through the IO controller (IOC); in step 2, cached copies are invalidated and data is written to main memory through the memory controller (MC); in step 3, the processor or the prefetch engine fetches data from main memory into the cache.

Figure 2.2 depicts the steps to move data from the network to a processor using cache injection. 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 decreased. Reducing memory traffic may translate into performance improvements for

memory-bound applications.

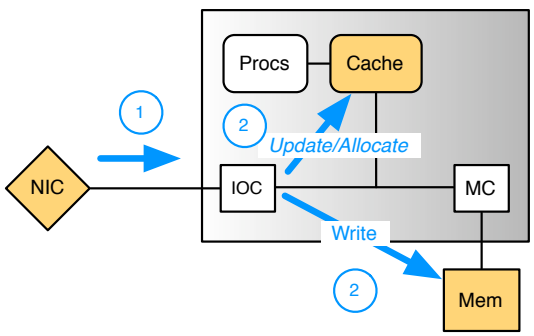


Figure 2.2: Cache injection operation initiated by the NIC. Unlike the memory write operation, step 2 allocates/updates incoming network data into the cache. If the processor uses this data promptly, there is no need to fetch it from main memory.

2.2 Cost considerations

The benefits of cache injection on application performance stem from reducing memory latency and memory pressure. Cache injection, however, presents costs that have to be considered. In this section, I analyze these costs. The base architecture in consideration is based on a Power5 architecture [54]. This system represents a modern multi-core system with an integrated, on-chip memory controller (see Figure 2.3).

Modern systems provide a hierarchy of caches, some local to a particular core, and others shared by a number of cores. One of these caches must be selected when using cache injection. Selecting a local cache provides lower latency at the expense of lower capacity. Targeting a local cache may require information from the OS about the location of the consumer of data. A simpler implementation of cache injection may target a shared cache. A shared cache provides greater capacity at the expense of higher latency. This greater capacity reduces the probability of evicting the application's working set.

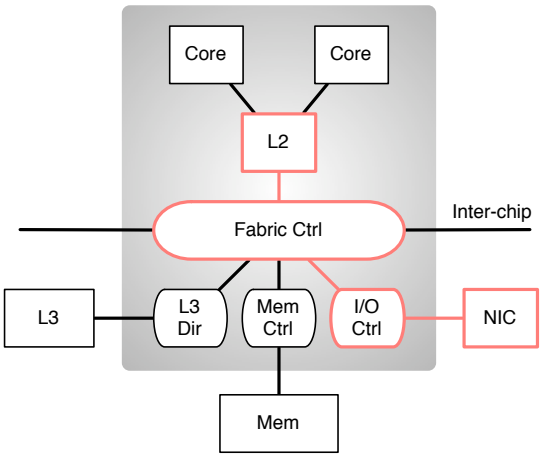


Figure 2.3: Base architecture for cache injection (based on IBM's Power5).

Major architectural modifications to the individual caches are not needed to implement cache injection. The caches in the base architecture are snooping associative caches and implement an extension of the MESI coherency protocol. When cache injection is enabled, data is written to main memory and allocated into the cache. The state of the appropriate cache block is set to clean-exclusive. If cache injection only writes to the cache, the state of the block can be set to modified-exclusive. Thus, when it is evicted from the cache, it will be written back to main memory. Both of these states are part of the existing coherency protocol. Cache replacement policies remain the same (pseudo LRU). Also, caches are expected to be at least 8-way associative [28] to reduce evictions of recently injected data. The base architecture implements a 10-way L2 cache and a 12-way set associative L3 cache.

As shown in Figure 2.3, the base architecture already provides data and control paths to the cache through the fabric controller. In the base architecture, the NIC issues invalidation requests to the cache and data requests to main memory when moving incoming network data to the host. The fabric controller forwards the invalidation requests to the cache and the data requests to the memory controller. Cache injection replaces invalidation requests

with allocate requests. Thus, cache injection does not need any additional control or data paths.

The major modifications when enabling cache injection are implemented on the NIC. Each transaction issued by the NIC adds an identifier of the target device. In other words, the NIC determines the data's destination (L2, L3 or main memory) using this identifier. The chipset should be able to route transactions from the NIC to the specified device. Also, the NIC's firmware has to be modified to implement a particular injection policy. An injection policy determines when and what to inject into the cache. These policies, as shown later in this chapter, may require information from the OS, the application, and the compiler. The communication library will also require modifications to communicate this information to the NIC. The operating system and applications, however, can run unmodified.

In summary, minor architectural changes are necessary to enable cache injection in modern architectures. The NIC, however, requires sufficient resources to implement the injection policy of interest and may also need information from the host that can be provided by the communication library. The OS and applications can run unmodified.

2.3 Comparing cache injection and data prefetching

In the previous section, I analyzed the architectural and system costs of implementing cache injection. In this section, I provide the context for this technique by comparing it with data prefetching. Prefetching is a well-know, widely implemented technique to hide memory latency. Using simulation and a micro-benchmark, I show that cache injection, unlike prefetching, can reduce memory traffic due to network data. This reduction in memory bandwidth usage is significant for applications whose memory bus is already saturated.

Cache injection and data prefetching strive to reduce data 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—initiated by the application, the OS, or the compiler, while cache injection is producer-driven—initiated by an I/O device, e.g., a NIC.

Many studies have shown that prefetching can be an effective technique to reduce data latency. However, prefetching has a significant disadvantage when compared to cache injection. Prefetching data from I/O devices consumes memory bandwidth due to two transactions: (1) transfer of data from the I/O producer to memory (while invalidating cached copies); and (2) fetching data from memory when demanded by the consumer. With cache injection, the second transaction is not necessary (assuming that the data is used promptly), decreasing the amount of data that has to go over the memory bus.

Both techniques may not perform optimally. Certain applications may not use sufficient computation instructions in between memory accesses to allow prefetching to hide memory latency. Both techniques are prone to cache pollution if data brought to the cache is not used promptly (see Section 2.4). Table 2.1 summarizes the differences between prefetching and cache injection.

Table 2.1: Prefetching vs. cache injection.

	Prefetching	**Cache injection**
Resources	1) write to memory 2) fetch to cache	1) write to cache
	use memory bw	reduce bw usage
	reduce data latency	
Fails when	data is not used promptly	
Applicability	general-purpose	limited to I/O
Type	consumer-driven	producer-driven

In the next two sections, I provide experimental results comparing cache injection and data prefetching. Section 2.3.1 describes the testing environment, while Section 2.3.2 describes the methodology and results.

2.3.1 Experimental framework

In this section, I describe the infrastructure to compare, experimentally, cache injection and data prefetching. Since no architecture exists that implements cache injection, I use simulation.

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 [7]. Mambo has been extended with an implementation of cache injection to the L3 cache [6]. The simulated machine is based on a cache-coherent, distributed shared memory architecture.

The high-performance communication system consists of a simulated high-performance NIC that attaches to Mambo and an OS-bypass zero-copy network stack [18]. The NIC is capable of running arbitrary functionality. It interacts with the host system through conventional write-invalidate memory operations and through non-binding write-allocate/update cache injection operations.

The network stack provides an unreliable datagram connectionless service with a UDP-like interface. I implemented this interface using an OS-bypass, zero-copy design which is common in high-performance networks. This implementation, Fast UDP [17], consists of code running on the host and code running on the NIC. The code running on the host is a user-level library that virtualizes NIC resources to applications. The code running on the NIC implements message matching and checksum processing.

In commodity UDP implementations, when a packet arrives from the network, the NIC copies the message to a kernel buffer and raises an interrupt to notify the kernel about its arrival. The OS processes the packet through the UDP/IP stack and then copies the payload to user space. In Fast UDP, the NIC has been instrumented to partially process UDP packets so that the payload is transferred directly from the NIC to user space, while the header (control information) is copied to a kernel buffer. Thus, the kernel remains aware

of incoming network packets, but does not incur overhead of processing application's data (including an extra copy to user space).

Like UDP, message matching semantics of Fast UDP are based on an IP address and a port. Unlike conventional UDP implementations, this operation in Fast UDP is performed on the NIC. The information about receive UDP buffers is shared by the OS with the NIC when a user posts a UDP receive. When a UDP packet arrives from the network, the NIC matches the packet using its destination port, and if a user has posted a receive for that port, the payload will be delivered to the user buffer. UDP checksum on the packet is performed on the NIC to avoid the transfer of erroneous data to the user.

2.3.2 Experimental evaluation

Using the infrastructure described in the previous section, I compare quantitatively cache injection and data prefetching by measuring memory bandwidth and execution time of a micro-benchmark in three configurations: (1) base case with no optimizations; (2) base configuration with prefetching; and (3) base configuration with cache injection. The micro-benchmark performs a linear traversal of incoming network data in calculating a reduction operation. This micro-benchmark represents a stage of computation that is limited by memory bandwidth and provides an optimal case for prefetching (linear traversal of data).

The machine configuration for these experiments is shown in Table 2.2. The simulated machine is based on a Power5 architecture [54] with a non-binding cache injection implementation to the L3 cache. The processor chip includes an L2 cache, a memory controller, and an I/O controller. The L3 cache is a victim cache [24] 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. The operating system is IBM's K42 research kernel [3].

Table 2.2: System configuration parameters.

Simulator	Mambo PowerPC full-system simulator
OS	K42
Transport	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, I measure the memory bandwidth used by the micro-benchmark in terms of the number of memory reads issued to the memory controller. As shown in Figure 2.4, 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 reduces the number of memory reads by up to 96% as all application accesses to incoming network data hit the L3 cache.

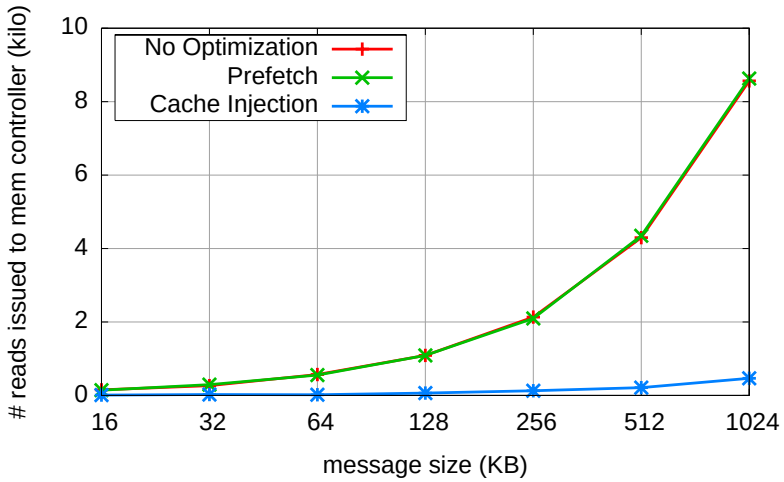


Figure 2.4: Memory bandwidth utilization.

Second, I measure the execution time of the application in processor cycles. As shown

in Figure 2.5, cache injection and prefetching outperform the base case as they both reduce the number of cache misses on network data. 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 the cache injection implementation targets the L3 cache [13, 14, 12].

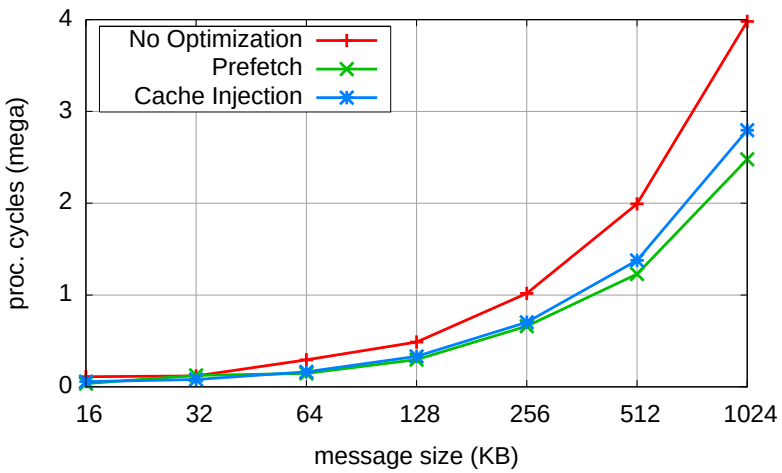


Figure 2.5: Execution time.

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 [9], a memory intensive application, at 384 tasks spends almost 75% of the overall application aggregate time in communication operations [57].

In summary, cache injection reduces memory bandwidth utilization compared to data prefetching. The results presented in this section represent an upper bound on the benefits provided by cache injection.

2.4 Limitations of cache injection

In the previous section, I showed that cache injection can reduce memory pressure significantly. Also, previous work [6, 28] showed 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 and producer-driven nature of this technique, namely timely transfer and identifying the consumer of data. For example, data may be transferred too early before the consumer can fetch it from cache, or data may be written to the cache of a processor which is not executing the consumer thread. In the former, cache pollution may occur, and in the latter, depending on the architecture, transfer from one cache to another may incur higher overhead than writing to main memory.

Cache injection requires explicit knowledge about the identity of the consumer. The NIC, the producer of data, has to choose quickly between a set of potential consumers. Even in a uni-processor system, the choice between an L2 cache, L3 cache or main memory has to be made. In a multiprocessor system, a core/processor has to be chosen. An incorrect choice of the target may result in higher delays than a conventional system without cache injection.

I refer to the process of transferring data from the NIC to a processor's cache directly without any knowledge of the application's usage patterns nor the state of the system as blind cache injection. In the remaining of this section, I show an application based on the Jacobi method in which cache injection without an appropriate injection policy (i.e., blind injection) suggests loss of performance. This motivates the study of injection policies, which are directly related to the effectiveness of this technique.

2.4.1 The Jacobi method

The Jacobi method [23] is an iterative algorithm used to solve a partial differential equation called the Laplace equation. This method can be used, for example, in calculating the temperature of a body represented by a multidimensional grid. At each time step, the Jacobi method computes the temperature of all interior points or cells based on their neighbors' values. The algorithm continues to refine the temperature values until a specific threshold is reached. The boundary points are fixed and set by boundary conditions.

A simple parallel implementation of this algorithm on a two-dimensional grid (n, n), may partition the problem domain into sub-domains that can be assigned to individual processes. Figure 2.6 shows the problem domain for a specific process. 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 rows for a particular process, the values of the boundary rows from its preceding and following neighbors' processes are needed. This requires data exchange between processes. At each time step, each process sends the values of its first and last rows ($2N$ values) to its neighboring processes accordingly. In a message-passing communication paradigm such as MPI [21], an algorithm (per process per iteration) that overlaps computation and communication can be outlined as follows:

1. MPI_Isend row boundary interior cells
2. MPI_Irecv ghost cells
3. Calculate interior cell values
4. MPI_Wait for ghost cells to arrive
5. Calculate boundary interior cell values

The steps of this algorithm can be classified into communication and computation stages. Steps 1, 2 and 4 are communication steps, and steps 3 and 5 are computation steps.

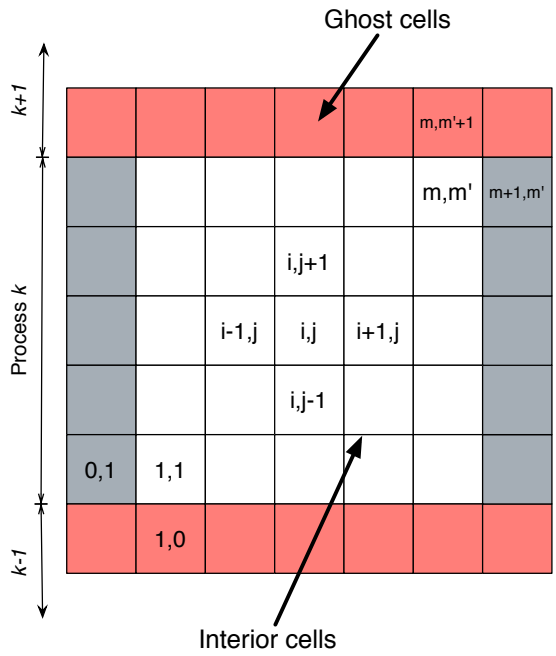


Figure 2.6: Jacobi problem domain for process k.

2.4.2 Performance using blind injection

Assume that the algorithm outlined above is running in a cluster of nodes, where one process runs in one node and each node is connected through a NIC to the cluster's network. At the beginning of execution, each process executes communication steps 1 and 2. Since the communication operations are non-blocking, each process continues executing step 3 even if the previous operations have not been completed. Also, assume that while executing step 3, ghost cells arrive to the NIC and are written to main memory (overlap of computation and communication). If step 3 is long enough for the communication stage to complete, then step 4 completes immediately, and the system continues to execute the last step. Otherwise, the process waits for the requested data to arrive and at that point moves to the last step.

In a system with blind cache injection, i.e., data is moved directly from the NIC to

a processor's cache as soon as it arrives, the overlap of communication and computation steps may 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 its working set again possibly replacing those blocks written by the NIC. After completion of step 3 and if 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 data 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 data latency and memory bandwidth benefits of this technique, injection policies to make informed decisions about when to inject data to the cache are necessary.

Two simple policies can overcome this problem: (1) for incoming network data whose 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 L3 caches, e.g., a Power5 machine contains a 36MB cache. The second policy requires OS information and may improve performance by speeding up the completion of step 4 (wait step).

Thus, if the consumer thread 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 informed injection policies. This information is based on the usage pattern of applications, the OS, the compiler and the communication library.

2.5 Injection policies

To leverage the performance improvements that can be provided by cache injection, adequate policies are needed. The goal of these policies is twofold: (1) to determine the appropriate place in the memory hierarchy for incoming network data; and (2) to identify the appropriate processor that will consume this data. In this section, I present a set of policies based on information from the OS, the compiler, and the application to make adequate decisions about what, when, and where to inject incoming network data.

Processor-direction Inject to the processor/cache where the consumer thread is executed. This information is provided by the OS and can be included in the memory descriptors that the NIC will use to match incoming messages. When a buffer is registered (pinned in memory) for communication, the OS adds the identifier of the consumer processor to the memory descriptor. MPI processes on a node are not expected to migrate. However, if they do, the OS can update the NIC with this information so that messages can be routed to the appropriate processor. This policy depends on the specific architecture where cache injection is implemented and it should take into consideration the latency for cache-to-cache transfers inside and outside a chip, and between chips as well as memory-to-cache latencies between chips and within a chip. Based on these parameters, the NIC can make a better decision to place incoming network data into the appropriate target to reduce memory bandwidth usage and data latency.

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

and/or communication library.

On-wait Inject to the target cache when the consumer thread is blocked waiting for incoming network data. The communication library may notify the NIC when the application is about to block waiting for network data. The notification may include a token identifying a particular memory descriptor on the NIC. When a matching message arrives from the network, the NIC writes to the cache if the receiving memory descriptor indicates that the application is waiting. Otherwise, data is written to main memory.

In-cache Inject to the target cache if the line is present. This policy requires querying the cache for a particular line or set of lines. Searches in a cache can be an expensive operation since the whole cache may be traversed. Smart searches [33] 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. The cost of this policy is directly related to the overhead of a cache search and is dependent on the architecture.

Meta-data Inject to the target cache the header or information about the payload of a message. In MPI, for example, the header is represented by the message envelope and may be used by the communication library shortly after written to the cache. The library may be polling a message notification queue, or it may be scheduled to run explicitly after a network interrupt issued by the NIC.

Size-dependent Inject to the L2 cache, L3 cache, or main memory depending on the size of the caches and the size of the incoming message. For medium or large messages, this policy may inject to the target cache the first part of a message and write the

rest to main memory. The idea is to let the prefetch engine fetch the rest of the message to the cache as it is needed. This policy reduces data latency by potentially overlapping computation on the first part of the message with memory latency for the rest of the message. In this case, there are no savings on memory bandwidth for the second part of the message.

The performance of these policies on application performance is dependent on several factors including the size and frequency of messages, the type of communication operations, and the ratio of communication to computation.

Chapter 3

Experimental infrastructure

In Chapter 2, I described the benefits and limitations of cache injection from first principles. I analyzed the architectural and system costs of implementing this technique; showed that cache injection reduces memory bandwidth utilization compared to prefetching using a micro-benchmark; demonstrated that cache injection can decrease application performance without an appropriate policy; and proposed policies to address the limitations of cache injection.

In this chapter, I describe the experimental infrastructure to demonstrate the effect of cache injection on application performance. The requirements of this infrastructure include cycle-accurate simulation of nodes (including cache injection), execution of unmodified message-passing applications at scale (hundreds of nodes), and functional and time-accurate simulation of the network. The resulting infrastructure is a distributed, cache injection cluster simulator that allows the execution of unmodified MPI applications at scale.

The study of cache injection at scale is part of a broader problem to simulate novel architectural features on many computational nodes. Researchers are exploring potential architectural changes for clusters, including a wide range of techniques such as hard-

ware matching support for scalable communication libraries [8] and radical architectural changes like processor-in-memory [53]. Unfortunately, the impact of such changes are difficult to predict analytically due to complex interactions between the architecture, operating system, system libraries, and applications.

Architectural simulators that examine the impact of system changes on application performance have not historically scaled well. For example, coarse-grained simulators skew dramatically when these changes are scaled up over tens or hundreds of systems. Similarly, cycle-accurate simulators, which can model each event to nanosecond accuracy in a single system, scale up poorly. Their running time increases dramatically even for a small number of processors. This limits designers in their ability to study how architectural changes affect scientific application performance as a cluster grows in scale.

To address this problem, I present an MPI-based cluster simulator designed to enable studies of architecture/operating system/application interactions on current and future architectures. Unlike previous work, my cluster simulator architecture uses existing clusters to simulate future clusters by coupling a cycle-accurate full-system node simulator[1] with an MPI-based high-performance network model. The resulting MPI-based cluster simulation system can be used to predict the impact of architectural changes such as cache injection.

In the following section, I describe the cluster simulator in detail followed by validation results showing that this simulator is accurate.

[1] A *full system* in this context means all the components that make up the compute engine of a cluster node: CPUs, caches, main memory, and the components that connect these parts.

3.1 Cluster simulator

To study the impact of cache injection on scientific applications at scale, I built an apparatus to simulate a cluster of machines based on a cycle-accurate simulator which leverages the parallel computation capabilities of current clusters. The simulated cluster is simulated on an actual cluster. A number of simulated nodes are run per physical node depending on the node's available resources. Communication between simulated nodes entails communication between physical nodes. By decoupling the simulation of individual nodes and the simulated interactions between them, a true parallel implementation can be achieved. The NIC and network model are distributed among the simulated nodes, resulting in a scalable simulator.

This apparatus consists of the following components: (1) a multiprocessor, multicore full-system simulator that simulates one computational node; (2) an OS, communication libraries and MPI applications for the simulated node; (3) a shim layer to bridge between a simulated node and a physical node; (4) a modeled NIC to communicate between simulated nodes; (5) a modeled network that models the network between simulated nodes; and (6) a message-passing communication system on the physical cluster that allows the launching and communication capabilities of the simulated cluster. I use the following terms for the rest of this chapter: a node refers to a physical host within the physical cluster; a machine refers to a full-system simulator (simulated node) running on a node; and a NIC refers to a modeled NIC that connects machines within the simulated cluster.

The simulated cluster is launched on a physical cluster by executing one instance of the full-system simulator per node. The machines communicate with each other over a modeled network through a modeled NIC. The NIC serves as a bridge between a (simulated) machine and a (physical) node. The NIC uses the existing messaging system (MPI) on the physical cluster to communicate with NICs running on other machines.

The full-system simulator provides a cycle-accurate simulation of a single-node, mul-

tiprocessor system. This machine provides a platform to study architectural features and configurations not yet available in current hardware. My goal in developing a cluster apparatus is not to create such single-node simulators, but to leverage their capabilities in a cluster setting. Communication between machines is achieved through a modeled NIC that can be loaded dynamically into the simulator and communicates with other machines through their associated NICs.

The specific simulator I use for this infrastructure is an augmented version of IBM's Mambo full-system simulator [7] with cache injection of incoming network messages [6]. The simulated machine is based on a Power5 architecture [54]. This architecture provides three levels of cache. The L1 and L2 caches are on board the processor chip, while the L3 cache is off-chip. The memory controller and L3 directory are also on board. The L2 is inclusive of the L1 cache, and the L3 is a victim cache (cache blocks evicted from the L2 are allocated in the L3 cache). Every machine runs the K42 research OS [3].

A simulated machine and its modeled NIC interact through a shim layer [18]. This layer provides a bidirectional path between the simulated machine and the NIC (see Figure 3.1). On one side, a machine communicates with its NIC through memory mapped registers. Using this mechanism, a user process can interact with the NIC directly, bypassing the OS and/or Hypervisor if needed. Access to these registers is controlled by the OS and/or Hypervisor. On the other side, a NIC communicates with its machine through a well-defined interface called the shim interface (see Table 3.1). This interface allows the NIC to write to the host's main memory and caches, to perform computation using the host's timing model, and to provide the appropriate functions to execute on accesses to memory mapped registers. In addition, the shim interface provides functions to load and unload network interface controllers and other devices at run time.

A modeled NIC connects its associated machine with the rest of the simulated machines running on the cluster. In addition, it functions as a bridge between its machine and the physical node it is running on. A NIC communicates with its associated machine

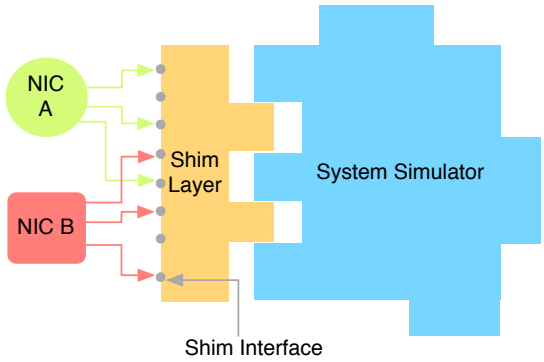


Figure 3.1: The shim layer provides the glue between the system simulator and the modeled NIC.

Table 3.1: A shim interface

Function	Description
`memory_read/write`	read/write to host memory
`cache_write`	write to L2/L3 cache
`schedule_job`	launch async task
`delay_cycles`	time delay on host
`raise_interrupt`	I/O interrupt
`memory_mapped_I/O`	functions to trigger on regs

through the shim layer and to other machines (running on other nodes) using the physical node's transport layer.

The NIC is capable of reading and writing to host memory and writing to the L2 and L3 caches. Writing to memory is performed by issuing write-invalidate bus transactions. Writing to a cache is performed in chunks of one cache block, and the state of the resulting block is set to clean exclusive [55]. Writes of less than one block are handled by a write with flush operation (flush the cache line first and then write the data into memory). Writes to a cache require the physical address of the destination to be block-aligned. Thus, writing incoming network data to a cache may involve writing the first few words using write with flush until the destination address is cache aligned, then writing full blocks to the cache.

Currently, all writes to the cache also update main memory under the assumption that most accesses to network data are read operations (no write-back operations are necessary). A different approach could be used for applications that frequently update incoming network data. Cache injection would only write to the cache, setting the state of incoming cache blocks to modified exclusive (dirty). When the values of these blocks are evicted, they are written back to memory.

The NIC uses the same timing model used by the machine simulator through the shim interface. For example, to perform a specific checksum on a particular packet, the NIC launches an asynchronous job (running in parallel with the simulated host) and assigns a particular delay (in terms of host processor cycles) to this task. Since every simulated machine runs asynchronously of each other, it is possible for one machine to move forward faster than others depending on the resources available for the simulator provided by its physical node. To set an upper bound on the time difference between any two simulated machines, the NIC synchronizes all simulated clocks every few tens of thousands of simulated cycles. The exact number is set based on the model of the network.

Each message sent between simulated machines is augmented by the NIC with a timestamp and delay information about the modeled network. The NIC then sends that information together with the machine's payload using the transport layer provided by the physical node. The receiving NIC waits to deliver the message to its simulated machine until its clock reaches the message's timestamp plus the modeled network delay [36]. This is possible because the network of the physical cluster appears lightning fast in comparison to the very slowly running simulated machines (in real time). The modeled NIC can deliver messages at any specified latency and bandwidth in simulated time, allowing the simulated cluster to use any type of network (physically possible or not). In this work, I use a Cray XT-3 network model based on Seshat [52], an execution-driven discrete event simulator to study application behavior under varying network characteristics.

The simulated cluster uses MPI to communicate between simulated machines. My

MPI implementation is based on MPICH [26] and MIAMI (Minimal Interface for An MPI Implementation). As shown in Table 3.2, the MIAMI API abstracts message-passing functionality into a small number of operations that can be implemented by the NIC. MPI operations are translated into MIAMI operations at the host communication library (see Figure 3.2). The library invokes MIAMI operations implemented by the NIC. The modeled NIC supports OS-bypass and allows for zero-copy MPI transfers.

Table 3.2: MIAMI API

Function	Description
`init`	initialize
`finalize`	clean up
`size`	number of processes in job
`rank`	my rank in job
`clock`	time in seconds
`tx_start`	start a send
`stx_start`	synchronous send
`tx_done`	check send completion
`rx_start`	post a receive
`rx_done`	check receive completion
`rx_probe`	probe for message arrival

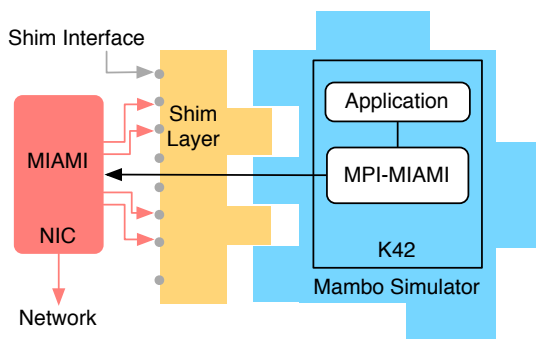


Figure 3.2: The implementation of MPI on a single simulated machine.

The communication between the simulated machine and its NIC through the MIAMI API is implemented with a user-level event queue. Each call to the MIAMI user library

generates an entry into this queue with the appropriate parameters. Once the event is written, a write to memory-mapped registers is performed to signal the NIC about this new event. The user queue is pinned in memory, cache block-aligned and each entry is one cache block in size.

To launch the simulated machines into the physical cluster and to communicate between nodes, I use the physical cluster's MPI library. The resulting MPI-based cluster simulation system is depicted in Figure 3.3 and its parameters shown in Table 3.3.

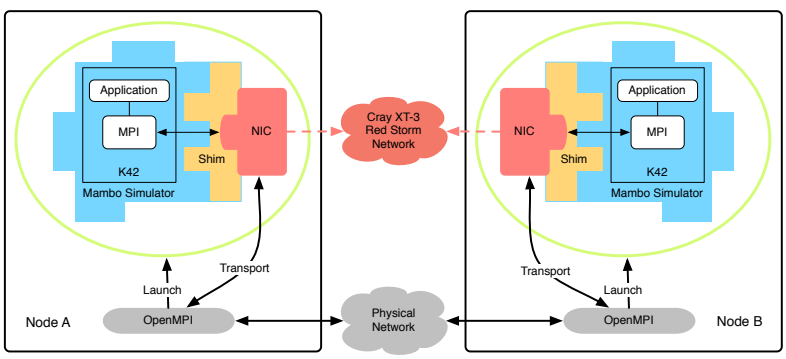


Figure 3.3: The parallel cluster simulator is simply an MPI job running on a physical cluster.

3.2 Validation

My cluster simulation system is based on a validated host simulator [7] and an accurate network model [52]. Even though I cannot validate this infrastructure against a real architecture, because such architecture does not exist, I provide evidence that this infrastructure is accurate.

I examine how the simulator could be used to study the effect of a simple architectural change (throttled network bandwidth) on cluster application performance and compare it

Table 3.3: Simulated system configuration

Feature	Configuration
Simulator	Mambo PowerPC full-system simulator
Architecture	Power5 with cache injection
Processor	1.65GHz frequency
Memory	825MHz on-chip controller, 275MHz DDR2
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
OS	K42
Comm. Lib.	MPICH-MIAMI w/OS-bypass & 0-copy
Network	Cray XT-3 Red Storm

against previous work [41, 16]. Because my system configuration is not identical to that of previous work, however, I cannot formally validate this result.

Network throttling is achieved by changing the parameters of the underlying modeled network. I also use these results to study how various simulation modes affect simulation times and how to verify that simulator behavior was consistent across different underlying execution platforms. I use two simulation modes: loose and accurate. In loose mode, the Mambo PowerPC simulator does not simulate the contents of the cache nor the behavior of the system using those caches which saves a lot of simulation time. In accurate mode, all the components of the system are simulated and, thereby, simulation time is slow.

In the following experiments, I use IS from the NAS parallel benchmark suite version 2.4 [5]. This code is a well-known integer sort benchmark. I chose IS because it was used in the previous work that I include for comparison. The runtime of IS is small, making it suitable for cycle-accurate simulations which take many thousand times longer to finish than the native execution time of the tested benchmark. In addition, I chose (small) NAS class A and B data sets for testing IS as opposed to the larger class C and D data sets for

the same reason.

Figure 3.4 is a comparison of IS on four nodes. I compare the results from running IS class A in loose and accurate modes and the class B results. I use the result of the unmodified bandwidth run as the 100% marker. I then plot the runs with fractions of that bandwidth and show the execution time IS reports as a percentage of the execution time when it is run at 100% bandwidth.

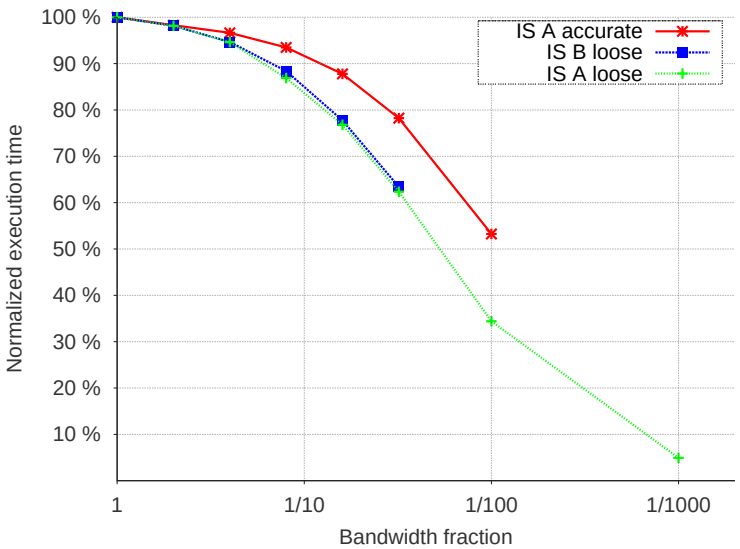


Figure 3.4: NAS IS benchmark on four nodes.

The plot lines in Figure 3.4 show very similar trends. The class A and class B runs show almost perfect overlap. The class A run in accurate mode is shifted to the right but shows a similar trend. I believe this shift is due to the changed CPU floating point performance to network bandwidth ratio when running in loose mode which alters the simulated processor performance.

In Figure 3.5, I repeat these IS simulations on sixteen nodes. I see a similar shift to the right of the accurate run versus the loose run as shown in Figure 3.4 for four nodes.

Figure 3.5 also includes the results from [15, 16]. In that work, the authors ran IS on

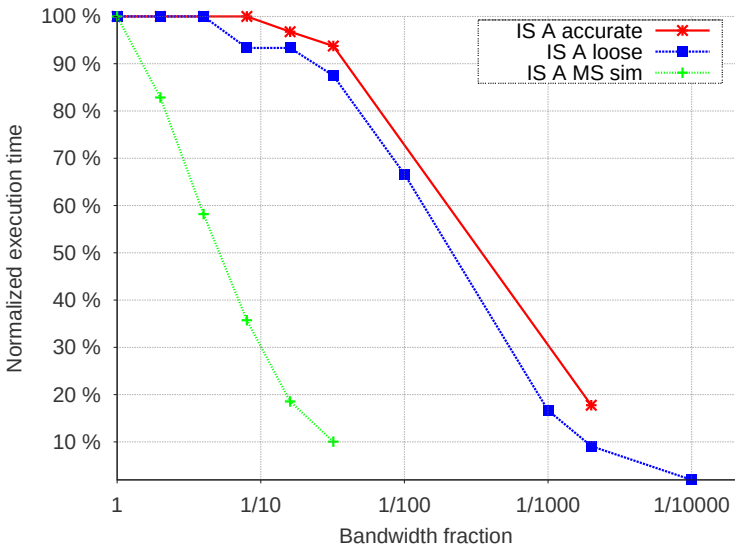


Figure 3.5: NAS IS benchmark on 16 nodes.

sixteen nodes and artificially slowed down the bandwidth of the network on a real system. This was done by introducing delays in the firmware of the Myrinet network cards in the cluster used for those experiments. I used the numbers obtained in those experiments and normalized them. The resulting line in Figure 3.5 does show a general similarity to the simulations ran on my simulator. Due to a different compute power to network bandwidth ratio, that line is much farther to the left. However, it does show about the same rate of decline in IS performance as the network bandwidth decreases.

These results combined with past work validating Mambo and Seshat show that the simulation infrastructure described in Section 3.1 can be used to predict how system changes would affect the performance of cluster applications. Also, I have shown elsewhere [19] that my simulator can accurately simulate a scalable machine by looking at a scalable application, and I show that it continues to scale when run inside my environment. My simulator has been able to scale to hundreds of nodes. In that work, I also demonstrated that the application's execution time starts to diverge as I increase the synchronization interval. As long as the synchronization interval does not exceed a specific

threshold (application dependent), applications perform deterministically. This behavior was also observed by previous work [20]. These, and other experiments, indicate that my simulator is accurate. Nonetheless, until I can simulate an existing system, direct validation cannot be done.

Chapter 4

Test environment

My evaluation of cache injection consists of executing MPI applications on a cache injection cluster simulator (see Section 3.1) over a variety of simulated system configurations. In this section, I describe the hardware and software environment used for this evaluation. This includes a description of the platform used for executing the cluster simulator, the simulated system base configuration, the cache injection policies implemented on the modeled NIC, the two parallel applications I ran on the cluster simulator, and the analysis tools used to gather application performance data and MPI collectives performance data.

4.1 Platform and simulated system configuration

I ran the cluster simulator on two cluster systems: Phoenix, a 16-node cluster at the University of New Mexico and Thunderbird, a much larger machine at Sandia National Laboratories. Phoenix is comprised of 16 compute nodes. Each node has two 2.20 GHz Intel Xeon processors and 1 GB of memory. A Gigabit Ethernet serves as the interconnect fabric. Phoenix uses the OpenMPI library [25] version 1.2.4 to transmit messages between nodes. The nodes run the Ubuntu Linux distribution using a 2.6.20 kernel. Thunderbird

is comprised of 4,480 compute nodes. They are dual 3.6GHz Intel EM64T processors with 6 GB of RAM. Thunderbird's network is an Infiniband fabric with a two level Clos topology. The nodes run Red Hat Enterprise Linux with a 2.6.9 kernel and use Lustre as the parallel file system. I used OpenMPI version 1.2.7 and OFED version 1.3.1 to connect to the Infiniband fabric.

The experimental results are gathered from a simulated cluster with very different characteristics than those systems on which the cluster simulator is executed. The simulated system configuration is shown in Table 3.3. This cluster is based on IBM eServer p550 nodes with cache injection running the K42 operating system. The nodes are interconnected with a Cray XT-3 Red Storm network.

4.2 Injection policies

In this section, I describe a subset of the policies from Section 2.5 that I implemented on the simulated system to evaluate cache injection. These policies are implemented on the NIC and place incoming network data and associated communication events into the appropriate level of the memory hierarchy (L2, L3 or main memory). The following policies are tailored for MPI and use the techniques outlined more generally in Section 2.5.

The first policy called **header** or **hl2** writes message headers (communication events) to the L2 cache. This policy is an instance of the meta-data and size-dependent policies, and is based on the interaction between the communication library (MIAMI) and the NIC. The host library and the NIC communicate through a queue of communication events residing on the host and pinned in memory. An event in the queue is used by the library to invoke a particular communication operation on the NIC. The library writes the operation's parameters to an event and signals the NIC about this operation using memory mapped registers. The NIC pulls the event from host memory, initiates the requested operation,

and writes a response to the appropriate event. When the headers policy is used, the response is written into the L2 cache. Since the user library polls the queue for responses from the NIC, the communication events are fetched directly from the L2 cache. The size of an event is exactly one cache block (128 bytes), and the user queue is block-aligned. The choice of L2 is appropriate because data is consumed promptly after it is injected and its size is small. To ensure the application's working set is not evicted from the cache, only one cache line is injected into the L2 cache per communication event.

Although I described a particular implementation of this meta-data policy, it can be generalized to other communication libraries where the NIC writes a small amount of data with information about a particular communication operation (header). Most architectures follow this model either by issuing an interrupt or by polling a queue. In my implementation, I use polling. Even though a similar policy has been explored by writing TCP/IP headers to the cache [28, 35], this work has been limited to two machines and to replaying traces on a single machine, lacking information about the global impact of cache injection on application performance.

The second policy called **payload** is an instance of the on-wait and application-dependent policies and injects application data (payload) into the L3 cache. The L3 cache is an appropriate target for application data due to its size (36 MB), the size of application messages, and to avoid polluting the L2 cache. The trade-off is a potentially higher data latency than the L2. This policy does not inject message payloads of more than half the size of the cache (18 MB) to avoid cache pollution.

To better control the injection of messages that are likely to be used quickly by the application, I further divide this policy into **preposted** (message has a matching receive) and **unexpected** messages. An application is more likely to use data sooner from a message matching a preposted receive than one that is unexpected. The preposted policy provides an approximation of the on-wait policy. Implementing an exact on-wait policy may involve communicating with the NIC frequently each time the application spins waiting for

a message. The communication library may infer the state of the application by identifying certain operations invoked by the application, e.g., MPI_Waitall. Passing hints to the NIC may be a costly operation since this information must cross the I/O bus. Thus, the preposted policy provides a reasonable trade-off of accuracy versus performance. The preposted and unexpected policies depend on the number of unexpected and preposted messages, e.g., if there are no unexpected messages, the unexpected policy has no effect.

Finally, the third policy called **message** or **hl2p** is simply the union of both the payload and header policies, i.e., inject communication events or headers to the L2 cache, and inject application payload to the L3 cache.

4.3 Parallel applications and performance analysis tools

In this section, I describe the applications and tools used to evaluate cache injection. All of these MPI codes were written in C, ran unmodified, and were built using a custom version of GNU's GCC 3.3.3 cross-compiler for PowerPC-64.

4.3.1 AMG from the Sequoia acceptance suite

AMG is an "algebraic multigrid solver for linear systems arising from problems on unstructured grids" [37]. It is one of several benchmarks used by Lawrence Livermore National Laboratory (LLNL) in its request for proposals and acceptance of the Sequoia supercomputer. Sequoia will be LLNL's next Advanced Simulation and Computing (ASC) machine.

The communication and computation patterns of AMG exhibit the surface-to-volume relationship common to many parallel scientific codes [27]. I chose AMG because it is a communication-intensive application which can, for large problem sizes, spend 90% of its

execution time using the MPI library to transfer messages. AMG uses collective operations and point-to-point messages of relatively small size (2 – 10 KB) [37].

AMG provides several solvers that can be selected from the command line. The data presented in this document is from solver 0 (the default—PCG solver). I chose this solver because it appears to place more demand on the memory subsystem than the other solvers. This solver may have better sensitivity to cache injection than the others available in AMG.

AMG has three distinct phases of operation. The solver runs in the third phase (solve phase), while the first two phases are used for problem setup. I augmented AMG to run in fast-forward mode (without simulating the caches) for the first two phases and enabled cache simulation before the third phase. Cache simulation is started hundreds of thousands of cycles before the solve phase to allow the caches to warm-up. Since the phase of interest is the solve phase, this allows me to advance the simulation quickly and to enable fully-accurate simulation just for the phase of interest.

I ran this code in weak-scaling mode with the default problem (a Laplace-type problem on an unstructured domain with an anisotropy in one part), setting the refinement factor for the grid on each processor in each direction to 1 (rx = ry = rz = 1).

4.3.2 FFT from the HPC Challenge benchmark suite

HPCC's FFT is one of seven benchmarks designed to examine the performance of HPC architectures using kernels with more challenging memory access patterns than previous benchmarks (HPL). These benchmarks were designed to bound the performance of many real applications as a function of memory access characteristics, such as spatial and temporal locality [40].

FFT (Fast Fourier Transform) measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT). I chose FFT be-

cause it requires all-to-all communication and stresses inter-processor communication of large messages. FFT runs in weak-scaling mode maintaining the same amount of work per processor. Each processor computed a vector of size 65536.

4.3.3 mpiP: an MPI profiling library

mpiP is a profiling library for MPI applications [56]. It gathers statistical information about point-to-point and collective operations. This information includes number, type and amount of time spent on communication operations, as well as message sizes. I use this tool to relate the performance of cache injection to the communication characteristics of an application. For a given application, I gather the communication to computation ratio, the sizes of the messages exchanged, as well as the communication primitives that consume most of the time in the MPI library.

4.3.4 IMB: Intel MPI benchmarks

IMB is a suite of benchmarks [29] designed to measure the performance of certain MPI communication operations. These operations include point-to-point and collective operations. I use these benchmarks to determine the impact of cache injection on individual collective operations. For example, if cache injection improves the performance of Allgather operations, then it is likely that applications spending a significant amount of time in these operations may show an improvement as well.

Chapter 5

Results and analysis

In this chapter, I describe the impact of cache injection on the performance of parallel applications. The performance of cache injection is a function of the memory and processor speed, cache injection policy, and the communication characteristics of applications. To determine application sensitivity to cache injection, I measure application performance in terms of execution time and floating-point operations per second (flops). Application performance is measured under a variety of configurations varying the factors of interest: memory and processor speed, injection policy, and application communication characteristics. To determine sensitivity to the last factor, I use two applications, AMG and FFT, whose communication signatures differ significantly. I conclude this chapter by associating cache injection performance with the communication parameters used in the LogGP model of parallel computation. The LogGP model provides a useful vocabulary to determine whether a particular application is sensitive to cache injection by solely analyzing its communication parameters. While this represents a useful approximation, the benefits of cache injection cannot be fully expressed by this model because characteristics such as message locality cannot be represented.

5.1 Memory and processor speed

In this section, I show the impact of memory and processor speeds on the performance of cache injection in relation to application performance. The experiments consist of measuring AMG's running time using different cache injection policies, memory and processor speeds, and numbers of MPI processes.

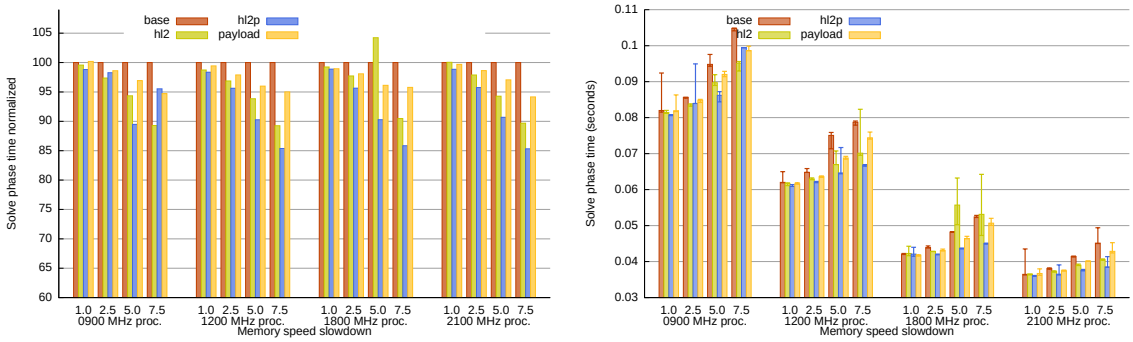
The baseline for the different memory speeds is based on a Power5 architecture as shown in Table 3.3. The memory speed is a function of the processor speed: the on-chip memory controller frequency is 1/2, and the DRAM DDR2 memory frequency is 1/6. A 2.1GHz processor has a 1050MHz (2100/2) memory controller and a 350MHz (2100/6) DRAM. The memory speeds considered for these experiments are shown in Table 5.1.

Table 5.1: Memory speeds as a function of processor speed.

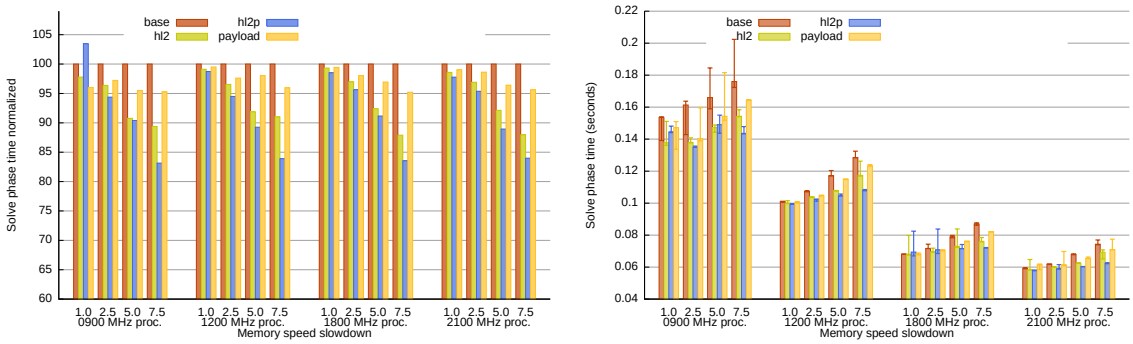
Slowdown	Controller	DRAM
1.0	1/2	1/6
2.5	1/5	1/15
5.0	1/10	1/30
7.5	1/15	1/50

I consider four cache injection policies: base, hl2, hl2p and payload. The base policy is the null hypothesis, i.e., no cache injection. Figure 5.1 shows the performance of AMG using different memory and processor speeds, numbers of processors, and injection policies. The graphs on the left show application performance normalized to cache injection's base case. The data plotted is the minimum of three runs. The graphs on the right show the median running time of three runs. As shown by the normalized graphs, the improvement provided by cache injection is inversely proportional to memory speed, i.e., the slower the memory system, the higher the benefit. This is the result of satisfying memory reads due to communication by the cache instead of main memory. The slower the memory system, the more expensive a cache miss becomes.

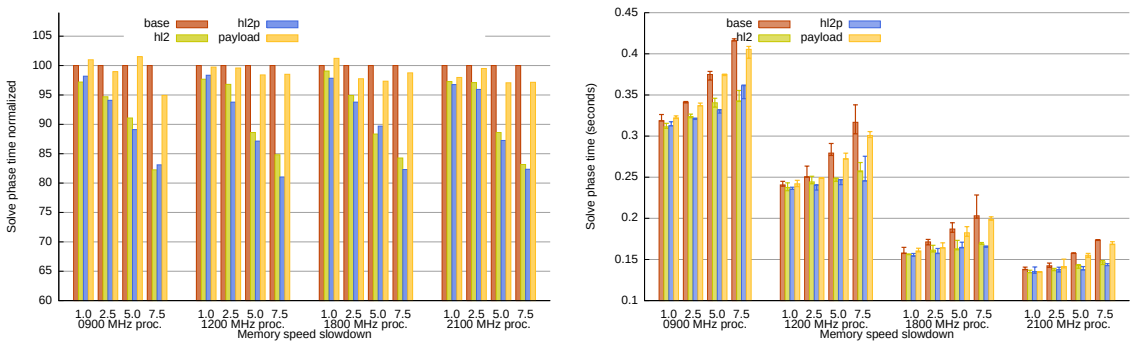
As demonstrated by the graphs on the right side, processor speed has a significant



(a) AMG's execution time at 8 nodes.



(b) AMG's execution time at 16 nodes.



(c) AMG's execution time at 64 nodes.

Figure 5.1: Performance of AMG solve phase as a function of memory and processor speed, cache injection policy, and number of processors. The graphs on the left show execution time normalized to cache injection's null hypothesis, i.e., no cache injection. The graphs on the right show execution time. Cache injection's improvement on performance is inversely proportional to memory speed.

impact on the performance of this application. This indicates that for this input problem size, the application is computationally-bound. However, processor speed does not play a significant role in the performance of cache injection (as shown by the normalized graphs). The reason is that memory speeds are fixed ratios of the processor speed. In other words, the ratio of processor to memory speed is constant across processors. For example, a 2.5 slowdown memory system for a 900Mhz processor has a 900/(2*2.5)Mhz memory controller, while a 2100Mhz processor has a 2100/(2*2.5)Mhz controller.

The important factor is not processor speed but the ratio of processor to memory speed (the memory wall in action). The higher the ratio, the higher the impact of cache injection on application performance. In other words, cache injection performance is directly proportional to the ratio of processor to memory speed. This result is significant even for multi-core architectures where the ratio of aggregated computational power to memory speed is high.

5.2 Cache injection policy

In this section, I show the impact of cache injection policy in the performance of AMG. The experiments consist of measuring the application's execution time using different cache injection policies, numbers of MPI processes, and memory and processor speeds. As discussed in the previous section, Figure 5.1 shows application performance improvement with hl2, hl2p, and payload policies. This improvement stems from a reduction of memory reads from the memory unit (memory controller and DRAM) due to communication events. In this section, I describe how each individual policy affects performance.

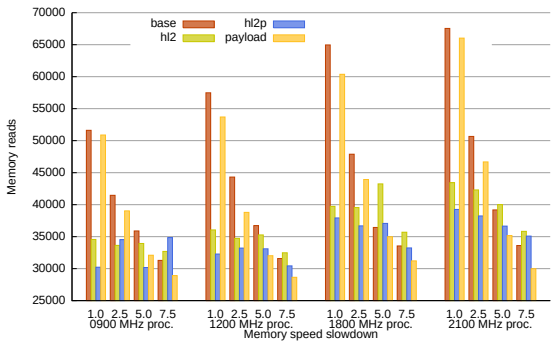
As described earlier, the NIC communicates to the host through memory using two types of writes: communication events (headers) and data (payload). Event writes are issued to the host event queue to notify the result of a specific communication operation

(small writes –one cache line); and data writes are issued to host buffers posted by the MPI application. The injection policies analyzed in this section are: hl2, write to the L2 cache all notifications to the host due to communication events (headers); payload, write application data (payload) to the L3 cache; and hl2p, write headers to the L2 cache and payload to the L3 cache. Data larger than half of the cache size is written to main memory for all policies.

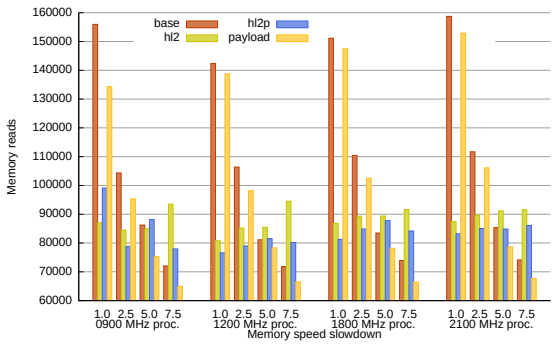
The hl2 policy (and therefore hl2p) can be very effective if there is a significant number of communication events because the host MPI library reads communication responses immediately after they are issued by the NIC (MPI library polls the event queue). Thus, the response may still be in the cache. The payload policy is independent of the communication event responses issued by the NIC. It relies on the amount of data received and the timely consumption of this data by the application. Since the application's data is not bounded to a particular size, data is injected to the L3 cache to avoid polluting the L2 cache. The hl2p policy combines both approaches by injecting headers and payload. This approach is useful when there is a significant number of communication events as well as when the application consumes its data promptly after arrival.

To understand the impact of these policies on application performance, I analyze the number of reads issued to the memory controller for each policy. Figure 5.2 shows memory reads as a function of the injection policy, memory and processor speeds, and the number of MPI processes. The hypothesis is that reducing the number of reads results in an overall improvement in application performance. However, this is not true for all cases. For example, Figure 5.2(c) shows a higher number of reads than the base case for the hl2 and hl2p policies and 5.0 and 7.5 memory speed slowdowns. But according to Figure 5.1(c), these policies, under the same memory configurations, actually improve performance. Another issue that needs to be addressed is that at low scale the number of memory reads decrease for hl2 and hl2p policies as a function of the memory slowdown, while at a higher scale the number of reads is independent of the memory slowdown factor.

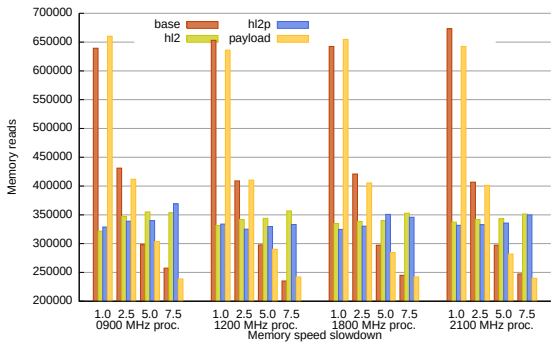
This is clearly expressed in the 2.1GHz processor data in figures 5.2(a) and 5.2(c).



(a) AMG memory reads at 8 nodes



(b) AMG memory reads at 16 nodes



(c) AMG memory reads at 64 nodes

Figure 5.2: AMG's number of memory reads carried out by the memory unit as a function of memory and processor speed, cache injection policy, and number of processors.

First, I address the higher number of reads for the header policies. An example may help clarify this issue. The MPI_Wait operation blocks until a particular communication event has completed, e.g., a receive event. The host implements the wait operation by continuously issuing requests to the NIC about the status of the event. The NIC writes a response into the appropriate entry in the host event queue. The host reads the response from the event queue. From the start to the completion of the wait operation, a number of memory reads are issued by the host to read the NIC responses. For the hl2 and hl2p policies, the host issues a higher number of memory reads than the base case because it reads the event notifications from the L2 cache and can issue the next request faster.

As shown in Figure 5.3, the number of event notifications to the host increases significantly for the hl2 and hl2p policies, especially at lower memory speeds. This graph also shows that the number of memory reads issued under the hl2 and hl2p policies are independent of the memory unit speed because most of the memory accesses hit the L2 cache. This also explains the higher number of memory reads in Figure 5.2 for low memory speeds, such as 5.0 and 7.5. For this application and input size, the number of memory reads is directly proportional to the communication events written by the NIC to the host.

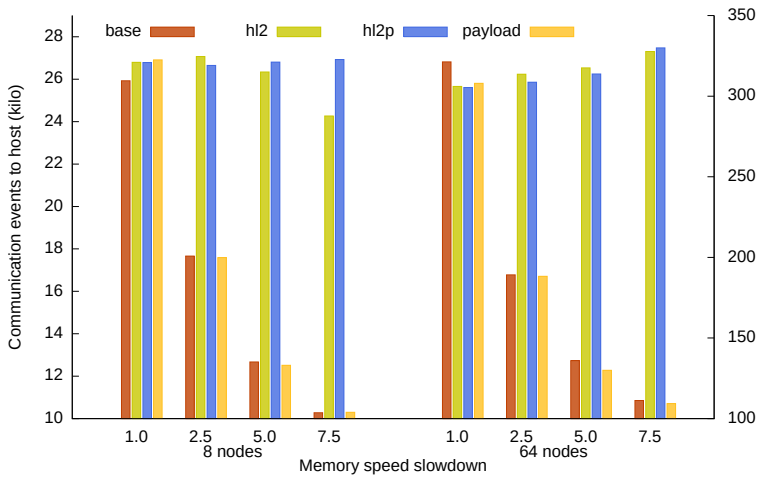


Figure 5.3: AMG's number of NIC to host communication events for a 2.1GHz processor as a function of memory speed and number of MPI processes. The left hand side uses the left 'y' axes and was gathered using 8 nodes; the right side uses the right 'y' axes and was gathered using 64 nodes.

The payload policy, as shown in Figure 5.2, behaves similar to the base case but with a lower number of memory reads. This reduction stems from cache hits on payload data. The decrease of memory reads across memory speeds (fewer reads with slower memory) for both the base case and payload policy can be explained using the wait example above. The number of reads the host can issue over a fixed period of time (waiting for the completion of a particular operation) decreases with lower memory speeds. As Figure 5.3 shows, the number of communication events is almost the same for both base and payload policies (communication events are written to main memory for both policies).

Second, I examine the issue related to the rate difference in memory reads to memory speed for low and high scale. This issue is demonstrated with a 2.1Ghz processor and header policies in figures 5.2(a) and (c). To better understand this issue, Figure 5.4 isolates these two graphs and provides a breakdown of the memory reads into useful and speculative reads. The memory reads issued to the memory controller are simply the number of reads carried out by the memory unit. Some of the reads may be speculative and their result might be discarded if the data is provided by a cache. As shown in the 8-node bars, the hl2 policy has a significant number (shown in red) of reads that are satisfied by the memory system, making this policy sensitive to the memory unit speed. In the 64-node case, most of the reads are provided by the cache and, thus, are independent of the memory unit speed. The decrease in the number of memory-satisfied reads stems from a higher percent of reads due to communication and a lower percent due to local data. Reads due to communication increase with the number of MPI processes due to the higher number of incoming packets.

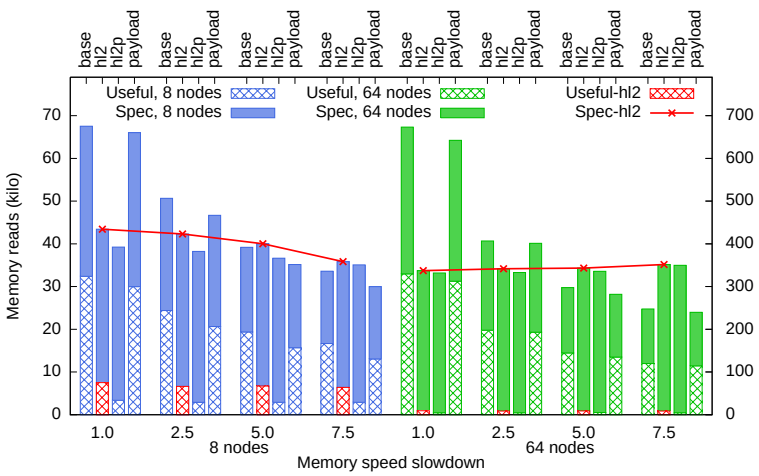


Figure 5.4: AMG's number of memory reads carried out by the memory unit for a 2.1GHz processor. Memory reads are broken down into speculative and useful reads. Data provided by a speculative read may be discarded if it is provided by a cache. The 64 node bars use the right 'y' axes, while the 8 node bars the left 'y' axes.

In summary, the headers policies perform well for applications issuing a significant

number of communication events. The data injected by this policy is expected to be read by the host promptly after it is written to the cache. Even though these policies may actually increase the number of reads issued to the memory controller, most of these new reads are satisfied by the cache. In other words, the reads satisfied by the memory unit are reduced. The header policies perform better than the payload policy due to the larger fraction of bytes generated by the communication events in comparison to the data used by the application. The payload policy improves application performance, indicating that the application consumes its data shortly after arrival from the network.

5.3 Application's communication characteristics

In this section, I show the impact of the communication characteristics of applications on the performance of cache injection in relation to application performance. To study this effect, I use two parallel applications, AMG and FFT, whose communication signatures are different (see Figure 5.6). The experiments consist of measuring application performance using different cache injection policies and numbers of MPI processes. The performance of AMG is measured, as before, in terms of execution time. The performance of FFT is measured in terms of gigaflops (one-billion floating-point operations per second), as specified by the HPC challenge benchmark suite.

From previous sections, I showed that the header policies improve the performance of AMG and provided higher improvement than the payload policy. The performance of these policies are actually dependent on the application, in particular its communication characteristics. As I will show, the impact of cache injection on the performance of FFT differs significantly. As shown by the left histogram of Figure 5.5, the hl2 policy provides no improvement, while the payload (and hl2p) policy improves performance between 6% and 8%. The right histogram shows the number of reads issued to the memory controller. This figure clearly shows that a reduction in memory reads results in performance im-

provement. The hl2 policy does not significantly reduce the reads to memory.

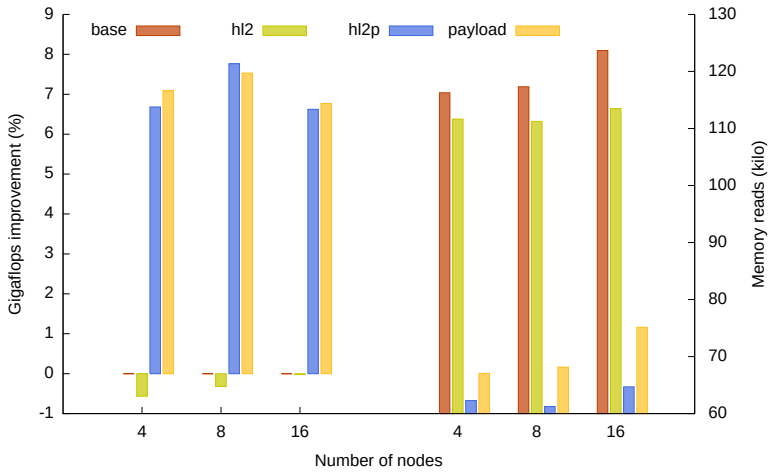
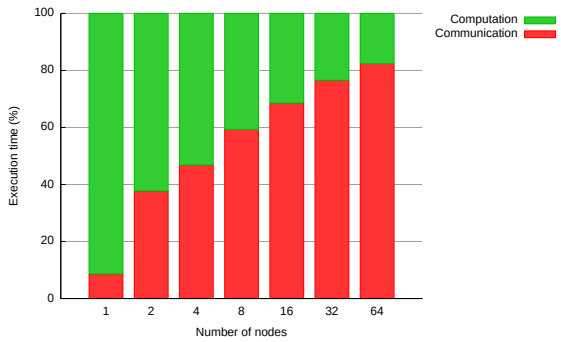


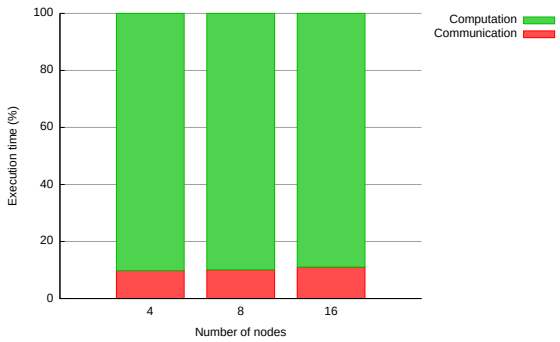
Figure 5.5: Impact of cache injection policies on the performance of HPCC's FFT. The histogram on the left show the gigaflops improvement (or degradation if negative) of the different injection policies. The histogram on the right show the number of memory reads issued to the memory controller. The performance improvement is inversely proportional to the number of reads.

The reduction in memory reads by the header policy stems from the communication characteristics of the application, in particular, the type of communication operations issued. As shown in Figure 5.6(c), AMG spends more than 20% of its communication time in MPI_Waitall operations. As mentioned before, wait operations result in continuous polling of NIC responses from memory. Reading these responses from the cache allow the host to process communication events faster. The more wait operations there are, the higher the improvement by the header policies. Thus, the performance of the header policies is directly related to the time an application spends in wait operations (test, wait, waitall, etc.). Other operations in the communication time of AMG (see Figure 5.6(c)), such as MPI_Allreduce and MPI_Allgather, do not affect the performance of cache injection since they use little time relative to the overall communication time as scale increases.

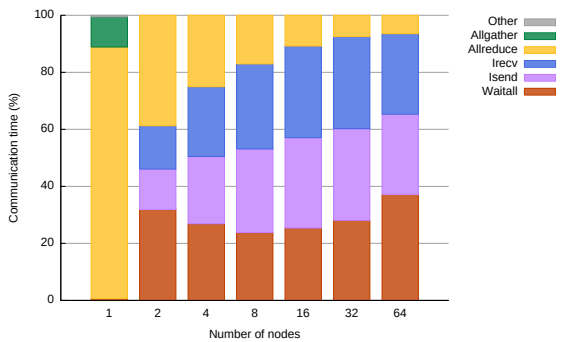
The payload policy, unlike the header policies, allows an application to access its communication messages faster (fetching them from the L3 cache rather than main memory).



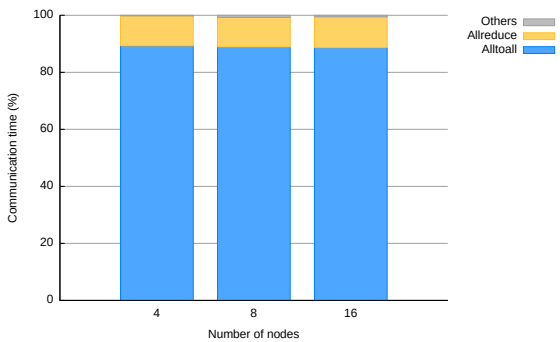
(a) AMG's comm. to computation ratio.



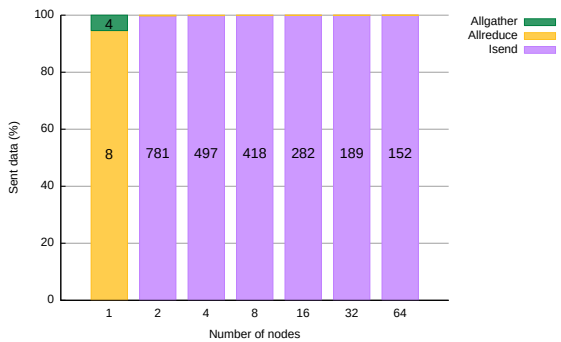
(b) FFT's comm. to computation ratio.



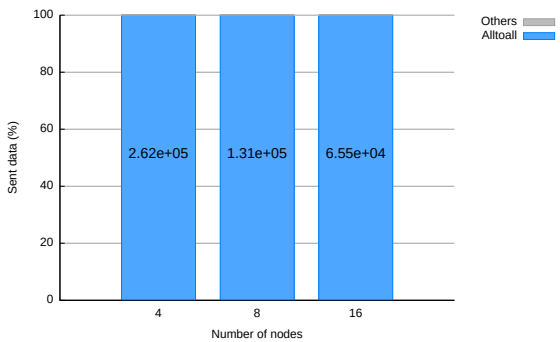
(c) AMG's communication operations.



(d) FFT's communication operations.



(e) AMG's sent data by comm. operation.



(f) FFT's sent data by comm. operation.

Figure 5.6: Communication characteristics of AMG and FFT. Figures (e) and (f) show the amount of data sent by each communication operation. The numbers inside the bars indicate the average message size sent (in bytes) by the appropriate operation. The difference in communication signature between AMG and FFT affect the performance of cache injection.

The improvement provided by this policy is related to the number of messages received and their size. As Figure 5.6(e) shows, most of the data sent by AMG uses point-to-point operations (MPI_Isend).

As mentioned above, the effect of cache injection is different for FFT. The hl2 policy does not provide any improvement since no wait operations are issued. Figures 5.6(d) and 5.6(e) show that more than 80% of the communication time is spent in MPI_Alltoall operations with sizes varying in the low hundred kilobytes. These operations provide a significant improvement in the performance of the payload policy. Even though there is only 6% to 8% improvement (see Figure 5.5) in the overall application performance, this is significant because only 10% of the application's time is spent in communication (see Figure 5.6(b)).

In summary, the performance of the different injection policies is dependent on the communication characteristics of the application: type of communication operations, message sizes, and communication to computation ratio. As exemplified by AMG and FFT, header policies showed positive sensitivity to wait operations while payload policies showed positive sensitivity to data intensive operations, such as MPI_Alltoall. In general, the performance of these policies depends on the ratio of meta-data to data of an application. In the AMG case, this ratio is high, most of the data written by the NIC is MPI's data. In the case of FFT, the ratio is low, most of the data written to the NIC is application's data. Another significant result is that unlike meta-data, which is expected to be read shortly after injection by the MPI library, application's data can also be read from the cache. In the rest of this section, I show how cache injection affects the performance of other collective operations.

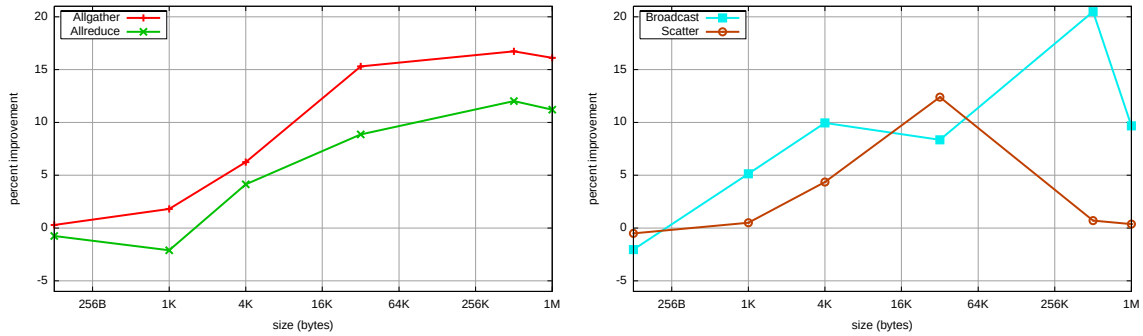
5.3.1 Collective operations

FFT showed positive sensitivity to cache injection largely because of the significant amount of time spent in collective operations, specifically MPI_Alltoall. In this section, I examine the effect of other collective operations on the performance of cache injection. This information can be used to identify other applications that may benefit from cache injection, and it aids in understanding the sensitivity of applications to cache injection based on the type of communication operations performed and the size of the messages exchanged.

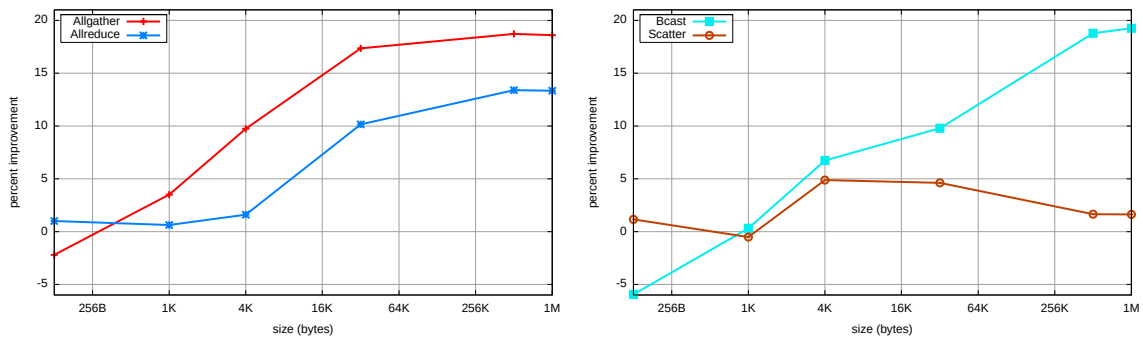
For this evaluation, I use Intel's MPI benchmarks (IMB). I run a selected subset of these benchmarks in two configurations: a null hypothesis or base case and an injection of the payload of MPI messages into the L3 cache. For each benchmark, I measure its execution time and use the average of 5 runs.

Results

Figure 5.7 shows the performance improvement of certain collective operations on 4 and 8-nodes. Allgather, Allreduce, Bcast, and Scatter were executed using a number of message sizes. There is a similar trend for both 4 and 8-node runs. Allgather, Allreduce and Bcast show overall improvements that increase with message size. The peak improvement of 20% is reached for messages of size around 512MB, where the increase in performance starts to flatten. The Scatter operation does not appear to gain sustained performance benefits with cache injection and, in some cases, show performance loss. The performance degradation shown for small message sizes is within the margin of error of these results. A source for performance degradation may also include evicting data from the application's working set.



(a) Sensitivity at 4 nodes.



(b) Sensitivity at 8 nodes.

Figure 5.7: Sensitivity of certain MPI collective operations to cache injection. Allgather, Allreduce, and Bcast show improvements up to around 20%, while Scatter does not seem to gain sustained benefits.

Analysis

To understand the sensitivity of these collective operations to cache injection, I examine the algorithms used in MPICH. The particular algorithm chosen for each collective depends mostly on the size of the communicator (n) and the size of the message. The version of MPICH used in this work implements collectives on top of point-to-point operations. The scatter operation uses a tree-based algorithm splitting data down the tree until the leaves are reached. The leaves receive only their piece, but intermediate nodes get more data than requested. Figure 5.8 shows an example of this algorithm for 8 nodes. The outbound degree per node is $O(log_2 n)$. It is highest at the root, $log_2 n$, and decreases

down the tree to 0 at the leaves. The inbound degree is $O(1)$, being 1 at the leaves and 2 everywhere else. Any benefit provided by cache injection can only be exercised during receive operations. Due to the small number of receives per node, the higher ratio of sends to receives per node, and the overlap of sends and receives, there is little to no gain provided by cache injection for this operation (see Figure 5.7).

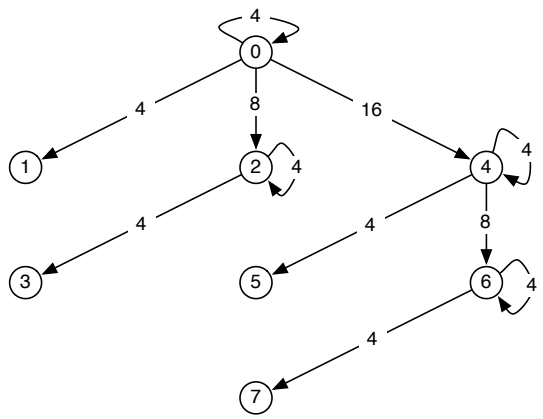


Figure 5.8: MPICH's scatter algorithm for 32KB and $n = 8$. The outbound degree per node is $O(log_2 n)$, while the inbound degree is $O(1)$ (at most 2). Message sizes along the edges are given in KB.

The broadcast algorithm is also tree-based, but unlike the scatter operation, the inbound degree per node is $O(log_2 n)$. Figure 5.9 shows an example of this algorithm on 8 nodes. Considering the number of incoming messages per node and the size of the messages, receive operations consume a significant portion of the overall time of this operation, thereby this algorithm shows positive sensitivity to cache injection. A similar argument can be made for Allgather and Allreduce operations.

In summary, collective operations affect the performance of cache injection. MPI_Allgather, MPI_Allreduce, and MPI_Bcast show performance increases as a function of message size, while MPI_Scatter does not provide sustained improvement. The performance improvement is related to the number and size of messages received at every node as specified by the operation's algorithm. Parallel applications with a significant communication

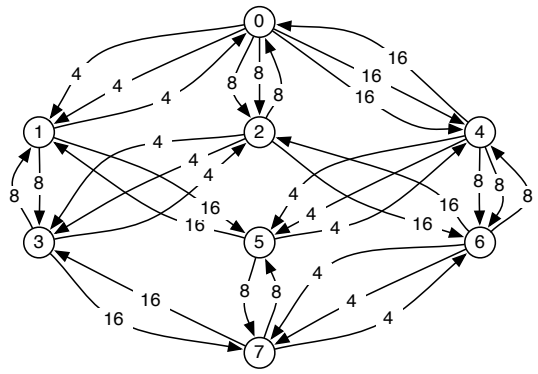


Figure 5.9: MPICH's broadcast algorithm for 32KB and $n = 8$. Both outbound and inbound degrees per node are $O(log_2 n)$. Message sizes along the edges are given in KB.

component spent on collective operations of sizes ranging from a few kilobytes to a few megabytes (depending on cache size) could potentially increase their performance using cache injection (payload policy).

5.4 Putting it all together: cache injection and the LogGP model

The results presented in this chapter showed how cache injection affects the performance of parallel applications based on the ratio of processor to memory speed, the injection policy, and the application's communication characteristics. In this section, I discuss the performance of cache injection in terms of a well-known model of parallel computation based on the results presented above.

The LogGP model of parallel computation [2], an extension of the LogP model [11], abstracts a parallel architecture into five parameters:

Latency (L) an upper bound on the time to transmit a short message from source to des-

tination.

overhead (o) the time that a host processor is engaged in sending or receiving a message and cannot do any other work. Since send and receive operations are often not symmetrical, I further consider send and receive overhead [15].

gap (g) the minimum time interval between consecutive message transmissions or consecutive message receptions at a node. The reciprocal of *g* corresponds to the available per-node communication bandwidth for short messages.

Gap (G) the gap per byte for long messages. The reciprocal of *G* corresponds to the available per-node communication bandwidth for long messages.

Processors (P) the number of processors.

The performance of parallel applications can be described using these parameters. Thus, to understand the impact of cache injection on application performance, I focus on studying the relationship between communication parameters and cache injection performance.

Cache injection, through header policies, improves the performance of latency–sensitive applications. Communication latency affects applications whose performance is dependent on timely arrival of short messages. Short messages are frequently used in synchronization operations such as barriers and all-reduce operations. All of the messages exchanged carry payloads of small size including no payload (zero-size). Also, other latency-sensitive applications may use point-to-point operations to synchronize with a subset of nodes (e.g., neighbor communication). The message exchange may occur by using asynchronous send/receive operations with their corresponding complete (wait) operations, or synchronous (blocking) operations. The performance of these operations is highly dependent on the communication library, since most of the message processing is done in the library. The involvement of the application is minimal because there is little

application data to process. In many instances, the involvement of the application is limited to receiving a notification from the communication library about the completion of a particular operation.

Cache injection, through the header policies, improves message processing in the communication library by reducing data latency on accessing incoming message headers. Data latency is reduced by an order of magnitude (tens of cycles vs. hundreds of cycles) by writing message headers to the L2 cache directly from the network. I showed an example of this improvement when analyzing the performance of cache injection on AMG. AMG (for the specific input size used in this study) uses small messages, and over 20% of its communication time is spent in wait operations. The header policies improved AMG performance due to the latency-sensitive nature of this application for this problem size.

Cache injection, through header policies, improves the performance of receive overhead–sensitive applications. As described above, the header policies improve the performance of the communication library by reducing the latency of accessing headers needed to process incoming network messages. This data latency is a significant component of the model's overhead parameter (the amount of time spent processing network data). The improvement provided by cache injection frees CPU cycles from network processing so that they can be used for the application's work. Examples of applications whose performance is dominated by communication overhead can be found in the literature [41, 16].

Cache injection, through payload policies, can improve the performance of Gap–sensitive applications. The Gap parameter represents the available communication bandwidth per node as a function of message size. In other words, applications whose performance is dominated by communication bandwidth are affected by this parameter. These applications exchange medium and large messages using point-to-point or collective operations. A common characteristic of these applications is that a significant amount of time in the processing of messages is spent in the application as opposed to the communication

library. This case is perfectly exemplified by high performance communication systems (such as the one shown in this work), where there is no memory copies from system communication buffers to application buffers. Application data flows directly from the NIC to end user memory. In these systems, the majority of the time is spent fetching message data from main memory to the cache.

Cache injection, through payload policies, improves message availability by writing message payloads directly to the L3 cache. This improvement can be leveraged by applications with high message temporal and spatial locality, i.e., applications that use incoming network data promptly after arrival and show spatial locality in using this data. Unfortunately, an application's message locality cannot be expressed using the LogGP model, thus cache injection can improve the performance of Gap–sensitive applications if they show message temporal and spatial locality. I showed an example of the effect of cache injection on this type of applications using the FFT benchmark. FFT's performance is known to be dominated by communication bandwidth [16]. This application uses mostly all-to-all operations of medium and long messages. The payload policies improve the performance of FFT by 7% even though it spends only 10% of its time in communication.

Cache injection may or may not improve the performance of applications as a function of the number of processors. The source of improvement or performance degradation of cache injection stems from incoming network communication. The higher the number of incoming network messages or the number of incoming network bytes, the higher the impact of cache injection on application performance. This impact may be related to the number of processors involved in communication depending on the application's communication characteristics. If an application uses a large number of collective operations, then increasing the number of processors/nodes will increase the amount of communication and, thus, the greater the impact of using cache injection. The extent to which cache injection affects performance for these applications is then dominated by the other communication parameters as discussed above.

Chapter 6

Related work

Cache injection is a technique to address the memory wall [58] specifically for I/O. Other techniques have been studied to address the memory wall from different perspectives. In this chapter, I describe this related work in the context of producer and consumer-driven techniques [10] to fetch data into a processor's cache. The consumer of data is a processor (on behalf of an application) while the producer can be any device that creates data for the consumer, for example I/O devices or other processors. In a consumer-driven approach, data is fetched when the consumer requests this data. For example, hardware prefetching starts fetching data, likely to be used by the consumer, based on the consumer's data access patterns. In a producer-driven approach, the producer sends data directly to the consumer under the assumption that the consumer will process it soon after arrival. For example, in cache injection, the NIC (producer) writes data into a processor's cache directly from the network.

In Section 6.1, I describe consumer-driven techniques to address the memory wall. Unlike cache injection, all of these incur memory latency and memory bandwidth usage for I/O communication. In Section 6.2, I describe producer-driven techniques for intra-processor communication. These techniques reduce data latency using data forwarding.

In data forwarding, both the consumer and the producer of data are processors within a computational node. In cache injection, these two reside in two different nodes connected through network interface controllers. The producer of data in cache injection can be thought of as either the NIC or a processor in a different computational node. Some ideas from data forwarding could potentially be used in cache injection taking into account the much higher latency between processors in different nodes. In Section 6.3, I describe producer-driven techniques for inter-processor communication. In particular, I describe previous work on cache injection. In this work, data is written into the cache from the I/O bus regardless of the state of the system. As shown here, this is prone to degradation of application performance. I build upon this work to overcome the shortcomings of cache injection by using policies to write data into the cache. Finally, in Section 6.4, I describe past architectures that precede cache injection for direct data transfer.

6.1 Consumer-driven techniques for the memory wall

Several techniques currently exist to manage the imbalance between processor and memory speeds. Data caching reduces memory latency for data access patterns exhibiting spatial or temporal locality. However, for computations with poor locality, caching does not help. Prefetching moves blocks of data into the cache before a processor's request. This technique overlaps memory latency with computation to improve processor performance. Hardware prefetching [4, 22] is based on usage patterns at run-time. The algorithms implemented by a prefetch engine have an overhead before detecting a particular pattern and cannot anticipate accesses of poor-locality computations. In software prefetching [44, 45], the compiler or the application inserts instructions into the code to start fetching data that will be used in the next few instructions. Software prefetching can only be used on architectures that provide prefetch instructions. This technique does not improve memory bandwidth.

Software access ordering [46] 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 [42] 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. In current architectures, accesses of I/O data incur in compulsory cache misses since data has to be fetched from main memory. Cache injection can reduce memory latency and memory bandwidth usage by placing data directly into cache from the I/O bus.

6.2 Producer-driven techniques for intra-processor communication

The Stanford Dash multiprocessor [39] included two producer-initiated operations update-write and deliver. The former writes data directly to all processors' caches that store the data, while the latter to a specific group of processors (cluster). In Poulsen's work [50], data is forwarded to other processor's caches to optimize shared accesses. Data forwarding is implemented by 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. IBM's POWER4 systems support cache-to-cache transfers (interventions) for all dirty lines and a subset of lines in the shared state [55].

Milenkovic [43] 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. The lprefetch and write_back instructions are inserted by the compiler. Unlike the studies mentioned above, the target architecture of Milenkovic's work uses a bus for interprocessor communication. This architecture has inherent scalability problems for more than a few processors. To my knowledge, the work by Milenkovic on bus based multiprocessors is the first to use the term *cache injection*. Milenkovic's preliminary data shows a reduction on cache miss ratio and bus traffic when using cache injection. The literature on producer-initiated mechanisms is extensive [10]. I outlined here only the most relevant for my work.

The difference between data forwarding and my 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 write data into the cache can potentially be applied to cache injection.

6.3 Producer-driven techniques for inter-processor communication

The work in the previous section focuses on reducing data latency for intra-processor communication within a computational node. The following studies focus on reducing data latency and memory bandwidth usage for inter-processor communication between computational nodes.

Bohrer et al. [6] proposed and analyzed cache injection as a mechanism to reduce data latency for incoming network TCP/IP packets. Using simulation and micro-benchmarks

for network and disk communication, the authors showed significant improvements in execution time. Huggahalli et al. [28] provided a more thorough study showing a significant reduction in both data latency and memory bandwidth usage for the benchmarks SPECWeb9, TPC-W and TPC-C. Their work also uses simulation, focuses only on TCP/IP, and their injection policy is to always inject the whole packet (header and payload) to the L2 cache. Huggahalli used the term Direct Cache Access (DCA) to deliver inbound network traffic directly to the cache.

A common characteristic of these two and unlike my work, is that data is always injected into the cache and does not adapt to the application's needs because the immediate consumer of this data is not the application itself. The processing of TCP/IP packets is a desirable target for this blind injection policy (see Section 2.4) since network processing occurs immediately after writing the packet to the cache (signaled by an external interrupt). A significant portion of the improvements provided by DCA stems from improving the copy of a packet's payload from a system buffer to a user one. However, in a high-performance environment the payload may be delivered directly to the user and without issuing costly interrupts. As shown by my work, cache injection does provide benefits for certain applications in this environment.

A more recent study by Kumar et al. [35], provides an initial evaluation of DCA on a real machine. They use an architecture based on Intel's I/O Acceleration Technology (I/OAT) [51] which provides an approximation to DCA called prefetch hint. With prefetch hint, data from the network is written to main memory, and a hint is passed to the current prefetch engine to start fetching data into the cache. Unlike cache injection, this technology does not reduce memory bandwidth usage (this reduction may be the most significant for memory-bound applications). Their study shows that prefetch hint reduces a significant amount of processor utilization for certain micro-benchmarks and is limited to experiments between two nodes directly connected. Their target environment is a data center where many applications may be running at once in the same node. Therefore, ap-

plying DCA to the processing of network packets may increase processor availability for other applications. In my work, I relate the effects of cache injection to the performance of the parallel application which consumes the incoming network data.

Khunjush et al. [32, 31] added a network cache and architectural extensions to manage it to avoid memory copies (from system to user buffers) due to network stack processing. Incoming network data of small size (a cache line) is written into the network cache, while other messages are written to main memory. When an unexpected message (in network cache) is bound to a particular MPI process, the tag is changed to point to the corresponding user buffer. Initially, the unexpected message points back to a system buffer. When the message in the network cache is evicted, the cache line is written back into the correct memory location. For preposted receives of small size, a cache line is allocated in the network cache pointing to the corresponding user buffer.

The authors gathered traces from a real system and replay them into one simulated node which implements their architectural modifications. The benefits of this approach are limited to small size packets with the additional overhead of implementing and searching another cache in the system. This study is also limited to the analysis of one node. Introducing an additional cache to the system avoids polluting the current caches but at a costly architectural change.

6.4 Past architectures for direct data transfer

A few distributed shared memory multiprocessor machines were constructed with the goal of reducing data latency due to interprocessor communication. The CM-5 machine [38] provided a logical connection between the network controller and the cache. Since there is no physical connection between these two, the processor is in charge of the data transfer, hindering its ability to overlap communication and computation.

The AP1000 machine [30] provided physical access between the network and the cache. Among several data transfers, it provided line sending which allowed data to be transfered directly from the cache to the network. The Alewife shared memory machine [34] allowed direct access to the underlying message-passing mechanism, providing both paradigms: shared memory and message-passing. This machine provided physical access between the network and the cache through a single-chip of communications and memory management unit (CMMU). Among other data transfers, it provided direct register-to-register transmission reducing data latency even more so than cache injection. However, the applicability of this type of operation is limited, as there are a few number of registers, and data should be consumed shortly after it is written.

Pakin's work [49] studied the impact of message traffic on memory performance for message-passing, distributed memory systems. Through simulation, he analyzed two configurations to logically connect the NIC with the memory system. The first configuration connected the NIC with main memory just as traditional systems do; the second connected the NIC to an off-chip cache as cache injection does. The first configuration used DMA to transfer data, allowing the processor to do work while messages are being transfered. The second configuration used the processor to transfer data. Unlike cache injection, this configuration does not have a physical connection between the NIC and the cache, but instead has the processor move data between these two.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this work, I present a study of the effect of cache injection on the performance of parallel applications. This effect is a function of the processor to memory speed ratio, the injection policy, and the application's communication characteristics. Unlike previous work, where injected data was used by the host network stack to reduce the overhead of memory copies, my results show that applications can directly use incoming network data injected to the cache. For example, the performance of HPCC's FFT improves by 8%. This improvement is significant since this application spends only 10% of its time in communication operations.

My results also show that the effect of cache injection on application performance is directly proportional to the ratio of processor to memory speed. This result makes cache injection a viable technique to improve the performance of applications dominated by the memory wall. This also suggests that cache injection can alleviate the imbalance of aggregated processing power to memory speed in multi-core architectures. I expect that cache injection with an appropriate policy, e.g., processor direction, can improve the

performance of communication- and data-intensive applications in these architectures.

The choice of injection policy is also an important factor in the performance of cache injection. For example, injecting communication meta-data information improved the performance of AMG, while injecting application's data improved the performance of FFT. The meta-data and application's data policies are not cache intrusive due to the small footprint of meta-data events (one cache line) injected to the L2 cache and the large size of the L3 cache to which the application's data was injected. The growing capacity of modern caches may reduce the intrusiveness of cache injection.

Application's communication characteristics greatly affect the performance of cache injection. For example, collective operations such as MPI_Alltoall, MPI_Allreduce, MPI_Bcast, and MPI_Allgather show positive sensitivity (up to 20% improvement) to cache injection as a function of message size. Collective operations benefit from cache injection due to the aggregated speed-ups in processing the appropriate data in a tree-like fashion. I expect that memory-bound applications with a significant amount of collective operations of medium message sizes can improve their performance significantly.

To conclude, cache injection is a viable technique to improve the performance of parallel scientific applications. This improvement is dependent on the application's sensitivity to the memory wall, the type of communication operations, and message size. Cache injection is likely to improve the performance of applications dominated by the memory wall and with significant use of medium and large size collective operations.

7.2 Future Work

There are several avenues of future work. First, I would like to investigate the performance of other policies outlined in Section 2.5, in particular, the processor-direction policy for multi-core architectures. While the results in this study suggest that these systems would

prove an ideal platform for cache injection (high ratio of aggregated processing power to memory speed), an actual study on these architectures would provide further insight.

Second, I would like to investigate dynamic policy reconfiguration: can *bad policies* be detected on-line and thus reconfigured? can the NIC use packet information to automatically choose a particular policy? what information does the NIC need to reconfigure and implement *good policies*?

Third, I am interested in analyzing a wider range of applications. While the applications used here show that my hypothesis is true, studying a wide variety of applications would allow me to characterize the impact of other communication characteristics on the performance of cache injection. Also, using a wide-range of problem sets and input sizes would further this characterization.

Fourth, I am also interested in studying the balance of flops to memory speed to network-bandwidth in multi-core systems. This type of architecture will become prevalent in the next-generation supercomputers and will significantly change the flops to network band- width ratio. I want to evaluate the impact of such an architecture on parallel application performance.

Last, I would like to use Intel's DCA (direct-cache-access) architecture to implement some of the injection policies I proposed in a real machine. Even though DCA does not reduce memory bandwidth utilization, it provides an approximation to cache injection (NIC prefetch hint) that is available now and can be used to improve the performance of applications.

References

[1] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram 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] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*, pages 95–105, June 1995.

[3] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42: an open-source Linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[4] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *ACM/IEEE conference on Supercomputing (SC'91)*, pages 176–186, Albuquerque, New Mexico, 1991. ACM.

[5] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995.

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

[7] Patrick Bohrer, Mootaz Elnozahy, Ahmed Gheith, Charles Lefurgy, Tarun Nakra, James Peterson, Ram Rajamony, Ron Rockhold, Hazim Shafi, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo – a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[8] Ron Brightwell, Trammell Hudson, Kevin Pedretti, and Keith Underwood. An accelerated implementation of portals on the cray seastar. In *Proceedings of the 2006 Cray Users' Group Annual Technical Conference*, Lugano, Switzerland, May 2006.

[9] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

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

[11] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '93)*, pages 1–12, San Diego, CA, May 1993.

[12] Edgar A. León, Kurt B. Ferreira, and Arthur B. Maccabe. Reducing the impact of the memory wall for I/O using cache injection. In *15th IEEE Symposium on High-Performance Interconnects (HOTI'07)*, Palo Alto, CA, August 2007.

[13] Edgar A. León and Arthur 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, November 2006.

[14] Edgar A. León and Arthur B. Maccabe. Comparing cache injection and data prefetching for I/O in chip-multiprocessors. In *EuroSys'07. Poster Session*, Lisbon, Portugal, March 2007.

[15] Edgar A. León, Arthur B. Maccabe, and Ron Brightwell. Instrumenting LogP parameters in GM: Implementation and validation. In *Workshop on High-Speed Local Networks (HSLN'02)*, pages 648–657, Tampa, FL, November 2002.

[16] Edgar A. León, Arthur B. Maccabe, and Ron Brightwell. An MPI tool to measure application sensitivity to variation in communication parameters. In *10th European PVM/MPI User's Group Meeting (EuroPVM/MPI'03)*, pages 108–111, Venice, Italy, September/October 2003.

[17] Edgar A. León and Michal Ostrowski. An infrastructure for network development. Proof of concept: Fast UDP. In *USENIX'05 Annual Technical Conference. Poster Session*, Anaheim, CA, April 2005.

[18] Edgar A. León and Michal 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, October 2005. ACM SIGOPS.

[19] Edgar A. León, Rolf Riesen, Arthur B. Maccabe, and Patrick G. Bridges. Instruction-level simulation of a cluster at scale. Submitted to the *International Conference on High-Performance Computing, Networking, Storage and Analysis (SC'09)*, November 2009.

[20] Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. An adaptive synchronization technique for parallel simulation of networked clusters. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*, Austin, TX, April 2008.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Knoxville, TN, 1994.

[22] John W. C. Fu and Janak H. Patel. Data prefetching in multiprocessor vector cache memories. In *18th International Symposium on Computer Architecture*, pages 54–63. ACM, 1991.

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

[24] Ben Gibbs, Balaji Atyam, Frank Berres, Bruno Blanchard, Lancelot Castillo, Pedro Coelho, Nicolas Guerin, Lei Liu, Cesar Diniz Maciel, Carlos Sosa, and Ravikiran Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbooks, second edition, 2005.

[25] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[26] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[27] Van Emden Henson and Ulrike Meier Yang. Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2000.

[28] Ram Huggahalli, Ravi Iyer, and Scott 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.

[29] Intel Corp. Intel MPI benchmarks 3.1. `http://www3.intel.com/cd/software/products/asmo-na/eng/219848.htm`, August 2008.

[30] Hiroaki Ishihata, Takeshi Horie, Satoshi Inano, Toshiyuki Shimizu, and Sadayuki Kato. An architecture of highly parallel computer AP1000. In *IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing (PACRIM'91)*, pages 13–16, 1991.

[31] Farshad Khunjush and Nikitas J. Dimopoulos. Lazy direct-to-cache transfer during receive operations in a message passing environment. In *3rd Conference on Computing Frontiers*, Ischia, Italy, May 2006.

[32] Farshad Khunjush and Nikitas J. Dimopoulos. Comparing direct-to-cache transfer policies to TCP/IP and M-VIA during receive operations in mpi environments. In *5th International Symposium on Parallel and Distributed Processing and Applications (ISPA'07)*, Niagara Falls, Canada, August 2007.

[33] Changkyu Kim, Doug Burger, and Stephen 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, October 2002.

[34] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *7th International Conference on Supercomputing*, pages 195–206, Tokyo, Japan, 1993. ACM Press.

[35] Amit Kumar and Ram Huggahalli. Impact of cache coherence protocols on the processing of network traffic. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*, pages 161–171, Chicago, IL, December 2007. IEEE Computer Society.

[36] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[37] Lawrence Livermore National Laboratory. ASC Sequoia benchmark codes. `https://asc.llnl.gov/sequoia/benchmarks/`, April 2008.

[38] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5 (extended abstract). In *4th ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, pages 272–285, San Diego, CA, 1992. ACM Press.

[39] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[40] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC challenge benchmark suite, March 2005.

[41] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA '97)*, pages 85–97, Denver, CO, June 1997.

[42] Sally A. McKee, Steven A. Moyer, and Wm. A. Wulf. Increasing memory bandwidth for vector computations. In *International Conference on Programming Languages and System Architectures*, pages 87–104, Zurich, Switzerland, March 1994.

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

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

[45] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 62–73, 1992.

[46] Steven A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, Department of Computer Science, University of Virginia, April 1993.

[47] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *IEEE International Symposium on Workload Characterization (IISWC'07)*, Boston, MA, September 2007.

[48] Richard C. Murphy and Peter M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, July 2007.

[49] Scott Dov Pakin. The impact of message traffic on multicomputer memory hierarchy performance. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.

[50] David K. Poulsen and Pen-Chung 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.

[51] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, November 2004.

[52] Rolf Riesen. A hybrid MPI simulator. In *IEEE International Conference on Cluster Computing (Cluster'06)*, Barcelona, Spain, September 2006.

[53] Arun Rodrigues, Richard Murphy, Ron Brightwell, and Keith D. Underwood. Enhancing NIC performance for MPI using processing-in-memory. In *Workshop on Communication Architectures for Clusters*, Denver, CO, April 2005.

[54] 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.

[55] Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, January 2002.

[56] Jeffrey S. Vetter and Michael O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, Snowbird, UT, July 2001.

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

[58] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 3(1):20–24, March 1995.