Assessing the Impact of Cache Injection on Parallel Application Performance

Edgar A. León
University of New Mexico
leon@cs.unm.edu

1 Introduction

The memory wall [13], the continuing disparity between processor and memory speeds, adversely affects the performance of many applications [8], particularly data intensive computations [11]. Cache injection addresses this disparity by placing incoming network data into a processor's cache directly from the I/O bus. The effectiveness of this technique on application performance is dependent on several factors including the ratio of processor to memory speed, the NIC's injection policy, and the application's communication characteristics. In this work, I show that the performance of cache injection is directly proportional to the ratio of processor to memory speed, i.e., the higher the memory wall, the higher the performance. This result suggests that cache injection can be particularly effective on multi-core architectures (increasing number of cores compared to available channels to memory). Unlike previous work (focused on reducing memory copies incurred by the network stack) [1, 5], I show that cache injection improves application performance by leveraging the application's temporal and spatial locality in accessing incoming network data. In addition, the application's communication characteristics are key to cache injection performance. For example, cache injection can improve the performance of certain collective operations by 20%.

Cache injection [1, 5, 3] is one of several techniques to alleviate the memory wall [9, 10, 8]. This technique reduces data latency and memory pressure (the number of requests issued to the memory controller per unit of time) by placing incoming network data directly into the cache. In current architectures, incoming network data is written to the system's main memory and cached copies are invalidated. Cache injection replaces invalidate operations with updates if the corresponding cache lines are present, or allocate operations if they are not.

In the next section, I describe how cache injection compares to a widely-used technique to reduce data latency, namely data prefetching. Unlike prefetching, cache injection significantly reduces memory pressure for I/O. Prefetching is driven by the access patterns of the processor (consumer of data), while cache injection is driven by the NIC (producer). This producer-initiated model makes cache injection prone to cache pollution. In Section 3, I show an example of this problem, and describe injection policies that determine what and when to inject into the cache to minimize pollution. In Section 4, I characterize application sensitivity to cache injection. In particular, I show that the performance of this technique is dependent on the degree to which systems are affected by the memory wall, the injection policy, and the application's communication characteristics. Finally, I conclude in Section 5.

2 Cache injection vs. prefetching

Cache injection and data prefetching strive to reduce data latency by moving data into the cache before it is needed. Table 1 compares both techniques. Prefetching has an important disadvantage: prefetching data from I/O devices incurs memory bandwidth due to two transactions: (1) transfer of data from the I/O producer to memory and invalidating cached copies; and (2) fetching data from memory to the consumer. With cache injection, the second transaction is not necessary, decreasing the amount of data that has to go over the memory bus. Both techniques may pollute the cache if the fetched/injected data is not used promptly. For 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.

Table 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

Using simulation [2, 4] and cache injection to the L3 cache [1], I compare quantitatively the two techniques by measuring memory bandwidth utilization and execution time of a micro-benchmark that 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). Memory bandwidth utilization is measured using the simulator's number of memory reads issued to the memory controller [2].

As shown in Figure 1, cache injection significantly reduces the number of reads to main memory by up to 96% as all application access to incoming network data hit the L3 cache. As shown in Figure 2, cache injection and prefetching outperform the base case as they both reduce the number of cache misses. Cache injection performs competitively with prefetching. More details of these experimental results can be found elsewhere [3].

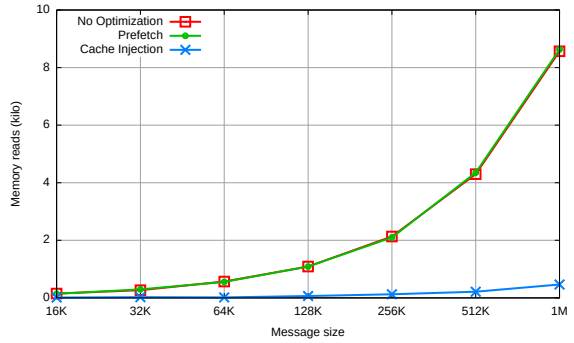

Figure 1: Memory bandwidth utilization.

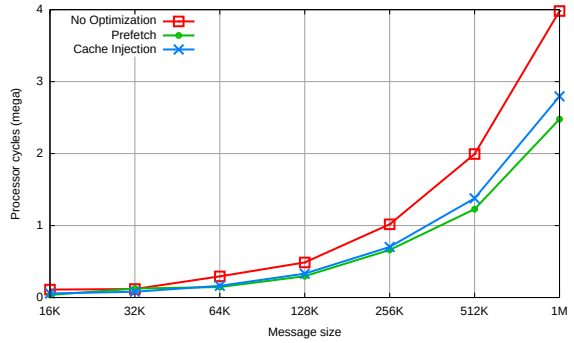

Figure 2: Execution time.

3 Injection policies

Cache injection presents challenges intrinsic to the producer-driven nature of this technique, namely timely transfer and identifying the consumer of data. With cache injection, data may be transferred too early for the consumer to fetch it off the cache, polluting and evicting useful data out of this resource. Consider the following MPI algorithm based on a Jacobi iteration:

1. MPI_Isend 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 Jacobi method is an iterative algorithm to compute the solutions of a system of linear equations and can be used, for example, to calculate the temperature of a body represented by a multidimensional grid.

If enough data is written to the cache from the NIC during step 3, the working set of the application may be evicted. The working set and incoming network data may be competing for space in the cache, and ultimately, when incoming network data does become part of the working set (step 5), it may not be in the cache anymore. This suggests that cache injection can decrease application performance if not used properly.

To leverage the performance improvements that can be provided by cache injection without polluting the cache, adequate policies are needed. The goal of these policies is to place incoming network data into the appropriate level of the memory hierarchy (L2, L3 or main memory). In a multiprocessor, multi-core system, the consumer processor/core also has to be identified. I present the following policies based on information from the OS, the communication library (MPI), the compiler and the application:

1. **Processor-direction**. Inject to the processor/core where the consumer thread is being executed. This information is provided by the OS and included in the memory descriptors that reside on the NIC to match incoming messages. MPI processes on a node are not expected to migrate.

2. **Compiler-driven**. Inject to the target cache when the application and/or compiler explicitly solicits the data.

3. **Headers (hl2)**. Inject to the L2 cache the envelope of a user message. This speeds up the communication library on the host. An envelope is usually small, 128 bytes (cache line size) in my case.

4. **Payload**. Inject to the L3 cache the data of a user message. This policy is further divided into **preposted** (message has a matching receive) and **unexpected**. An application is more likely to use data sooner from a message matching a preposted receive than one that is unexpected.

5. **Message (hl2p)**. Inject both headers and payload to the L2 and L3 caches respectively.

4 Evaluation

To analyze the effect of these policies on application performance, I created a scalable framework to simulate a cluster of cache injection systems. This framework combines an existing cycle-accurate node simulator (IBM's

Mambo [2]) with an MPI-based network model [12]. The resulting parallel simulator can be used to examine the effects of proposed architectural changes, e.g., cache injection, on cluster application performance. Also, it can execute unmodified MPI applications and is scalable to hundreds of simulated machines.

The simulated system configuration for this work is a cluster of Power5 with cache injection [1] machines interconnected with a Cray XT-3 Red Storm network. Each machine runs the K42 operating system and the MPICH-MIAMI implementation of MPI. This implementation leverages zero-copy and OS-bypass communication.

To characterize application sensitivity to cache injection, I implemented policies 3, 4 and 5 (see Section 3), and quantified their effect in the performance of AMG from the ASC Sequoia acceptance suite [6], and FFT from the HPC challenge benchmark suite [7]. AMG is a solver for linear systems arising from problems on unstructured grids. FFT measures the floating point rate of execution of complex one-dimensional DFTs.

As Figure 3 shows, the performance of cache injection is directly proportional to the ratio of processor to memory speed. This figure shows AMG's execution time as a function of memory speed slowdown for a variety of processor speeds. The important factor in the performance of cache injection is the ratio of processor to memory speed and not the absolute processor speed. The higher the memory wall, the higher the improvement on application performance. The performance improvements provided by the three injection policies studied here stem from a reduction in the number of memory reads served by the memory unit. In other words, cache injection increases the number of memory accesses satisfied by the cache. The header policies (hl2 and hl2p) provide better performance since 37% of the application's communication time is spent in MPI_Waitall operations. These operations create requests to the NIC checking for the result of a particular communication operation. NIC responses are written into the L2 cache under the header policies.

As shown by the left histogram of Figure 4, the payload policies (payload and hl2p) improve the performance of FFT by up to 8%, while the header policy does not provide any improvement. As shown by the right histogram in the graph, the performance improvement is inversely proportional to the number of reads issued to the memory controller, i.e. the lower pressure to memory, the higher the performance. For this application, the payload policies perform better than the header policy since 88% of the application's communication time is spent in MPI_Alltoall operations. This result also shows that cache injection can leverage the application's message temporal and spatial locality. The performance improvement for this application is significant since only 10% of the application's time is spent in communication operations.

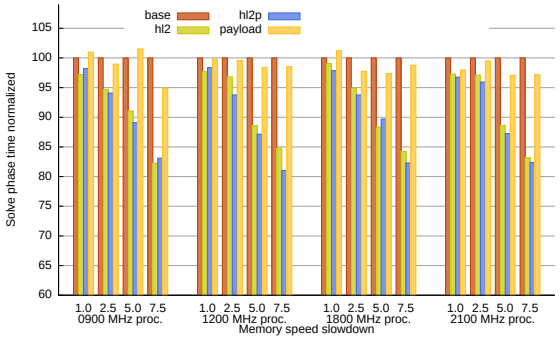

Figure 3: AMG performance normalized to base case as a function of memory and processor speed, and cache injection policy for 64 nodes.

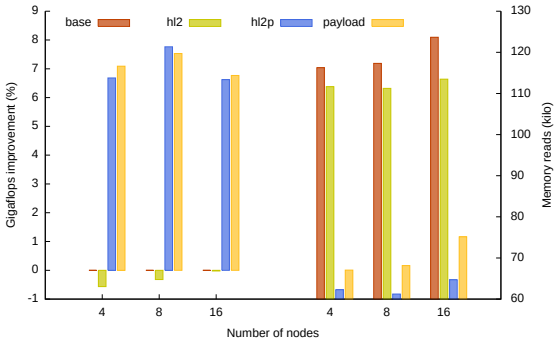

Figure 4: FFT performance as a function of injection policy, and number of nodes.

The different sensitivity shown by AMG and FFT to cache injection is a result of their different communication characteristics. As Figure 5 shows, cache injection improves the performance of collective operations such as MPI_Allgather, MPI_Allreduce, and MPI_Bcast by up to 20% as a function of message size. These improvements stem from a faster message availability at every level of the tree-based algorithms implemented by MPICH. MPI_Scatter does not provide sustained improvements due to the low number of incoming messages per node in the tree structure. This suggest that cache injection can improve the performance of applications using a significant amount of collective operations of medium to large sizes (as shown by FFT above).

5 Conclusions

Cache injection is a viable technique to improve the performance of parallel applications bound by the memory wall. To show this, I created a framework to study clusters of systems with novel architectural features at scale. Using this framework, I enabled cache injection to the L2 and L3 caches, and characterized application sensitivity using

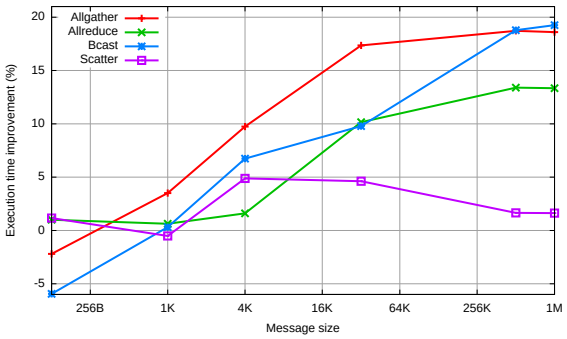

Figure 5: Effect of cache injection in the performance of collective operations as a function of message size.

a set of policies tailored for MPI. My results showed that the performance of cache injection is directly proportional to the ratio of processor to memory speed. This result suggests that the memory wall in many-core and multi-core architectures (increasing number of cores compared to available channels to memory) can be alleviated with cache injection.

I also show that the performance of cache injection is a function of the injection policy and the application's communication characteristics. Unlike previous work, I show that cache injection can improve application performance by leveraging the application's temporal and spatial locality in accessing incoming network data. In addition, cache injection improves the performance of collective operations such as Alltoall, Allgather, Allreduce, and Bcast by up to 20% as a function of message size. To conclude, cache injection addresses the memory wall for applications with low temporal locality and a significant number of collective operations of medium and large sizes.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.