Improving the Performance of Parallel Scientific Applications Using Cache Injection

Edgar A. León

Extended Abstract

The memory wall, the continuing disparity between processor and memory speeds, adversely affects the performance of memory bound applications, particularly parallel scientific computations. Cache injection addresses this disparity by placing data into a processor's cache directly from the I/O bus. The effectiveness of cache injection on application performance is dependent on several factors including timely usage of data, the amount of data, and the application's data usage patterns. To improve application performance, I present policies to place incoming network data into the appropriate level of the memory hierarchy (L2, L3 or main memory). Preliminary results from a set of policies tailored for MPI, show an overall improvement of 5% in the execution time of an actual application (AMG from the ASC Sequoia suite), and up to 15% improvement on specific stages of this code.

Cache injection and data prefetching strive to reduce memory 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.

Table 1: Prefetching vs. cache injection.

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

I compare quantitatively the two techniques by measuring memory bandwidth and execution time of a micro-benchmark using cache injection to the L3 cache [1]. 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).

As shown in Figure 1, cache injection significantly reduces the number of memory reads 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.

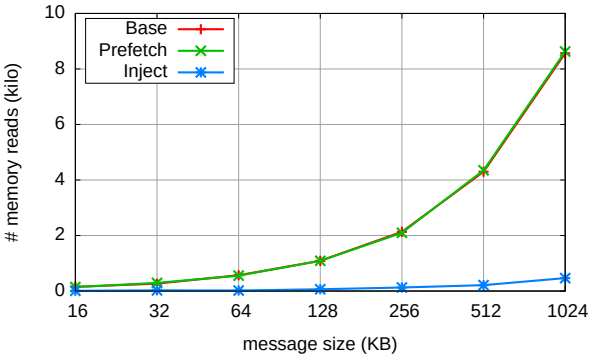

Figure 1: Memory bandwidth utilization.

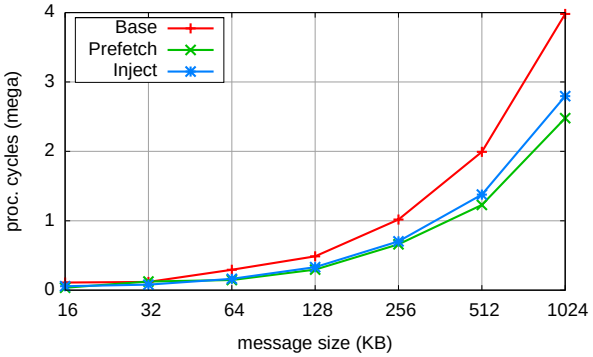

Figure 2: Execution time.

Cache injection presents challenges intrinsic to the explicit producer-driven nature of this technique, namely timely transfer and identifying the consumer of data. With cache injection, data may be transferred too early for the consumer to fetch it off the cache, polluting and evicting useful data out of this resource. Consider the following algorithm based on a Jacobi[1] iteration:

[1]The Jacobi method is an iterative algorithm to compute the solutions

1. MPI_Isend boundary data
2. MPI_Irecv boundary data
3. Perform local computation
4. MPI_Wait for remote data to arrive
5. Perform remote data computation

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**. 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**. Inject both headers and payload to the L2 and L3 caches respectively.

To analyze the effect of these policies on application performance, I created a scalable parallel cluster simulator by combining a cycle-accurate CPU simulator (IBM's Mambo) with an MPI-based network model. This simulator can be used to examine the effects of proposed architectural changes such as cache injection on cluster application performance.

of a system of linear equations.

The cluster simulator parameters for this work are: IBM's K42 research OS; Power5 node architecture with cache injection; Cray XT-3 Red Storm network between nodes; and MPICH-MIAMI with OS-bypass and zero-copy.

To provide initial evidence of the impact of these policies, I implemented policies 3, 4 and 5, and measured their effect in the execution time of AMG, a parallel algebraic multigrid solver, on 4 nodes. AMG's running time is divided into three phases: *SStruct*, *Setup* and *Solve*.

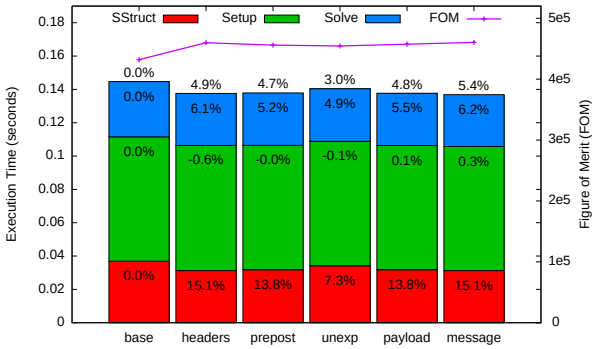

Figure 3: Performance of AMG with proposed policies.

As Figure 3 shows, the effectiveness of cache injection in reducing execution time varies from 0 to 15.1% depending on the phase of the computation and policy. In stages with no communication, cache injection cannot provide any benefit. The performance of cache injection is dependent on several factors including the computation to communication ratio. From the different policies, *headers* performs well as the MPI library uses a message envelope right after it is written into the cache. Overall application performance improvement varies from 3 to 5.4%. This figure also shows AMG's figure of merit which represents how fast a system performs under this benchmark (higher is better). All the policies presented here increase this figure.

These preliminary results are encouraging in showing that cache injection can improve the performance of parallel scientific applications with an appropriate policy. More work remains to be done including characterizing the types of applications that benefit from cache injection, and applying this technique to a variety of applications at scale under all of the proposed policies.

[1] 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.