Cache Injection for Parallel Applications

Edgar A. León
IBM Research, Austin*
eleon@us.ibm.com

Rolf Riesen
IBM Research, Ireland†
rolf.riesen@ie.ibm.com

Kurt B. Ferreira
Sandia National Laboratories‡
kbferre@sandia.gov

Arthur B. Maccabe
Oak Ridge National Laboratory
maccabeab@ornl.gov

ABSTRACT

For two decades, the memory wall has affected many applications in their ability to benefit from improvements in processor speed. Cache injection addresses this disparity for I/O by writing data into a processor's cache directly from the I/O bus. This technique reduces data latency and, unlike data prefetching, improves memory bandwidth utilization. These improvements are significant for data-intensive applications whose performance is dominated by compulsory cache misses.

We present an empirical evaluation of three injection policies and their effect on the performance of two parallel applications and several collective micro-benchmarks. We demonstrate that the effectiveness of cache injection on performance is a function of the communication characteristics of applications, the injection policy, the target cache, and the severity of the memory wall. For example, we show that injecting message payloads to the L3 cache can improve the performance of network-bandwidth limited applications. In addition, we show that cache injection improves the performance of several collective operations, but not all-to-all operations (implementation dependent). Our study shows negligible pollution to the target caches.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*design studies, modeling techniques*; C.1.4 [**Processor Architectures**]: Parallel Architectures—*distributed architectures*

General Terms

Performance, Experimentation, Design

Keywords

Cache injection, memory wall

1. INTRODUCTION

For 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 [30]. This disparity known as the memory wall [36] makes memory speed the limiting system performance factor for many types of applications. Even systems with infinitely sized caches are affected due to compulsory cache misses on data with poor temporal locality. Current techniques such as 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 [24]. Recent high-end, real-world scientific applications strongly depend on the memory characteristics of a system [29, 28]. 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 bandwidth and latency.

The advent of multi- and many-core architectures poses an even greater challenge to mitigate the memory wall. As shown in Figure 1, the theoretical memory bandwidth per core decreases with newer generation chips which contain more cores. Table 1 shows the details of the processors used for this figure [3, 11, 2, 18, 1]. We quantify the memory wall using *bytes per cycle per core* to take into consideration the absolute memory bandwidth per chip, the number of cores, and the processor frequency. The absolute memory bandwidth (GB/s) per core is not informative enough since it does not account for processor frequency.

The number of applications affected and the severity of the memory wall will increase in the near future as the number of cores increases in next generation processors. Manufacturers are making an effort to increase the number of memory ports and the memory bandwidth per bus but are hindered by pin count and other physical limitations. This means that the

*This work was partially supported by an Intel fellowship and an IBM grant under subcontract from IBM on DARPA contract NBCH30390004.

†Rolf Riesen participated in this work while working for Sandia National Laboratories.

‡Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

HPDC'11, June 8–11, 2011, San Jose, California, USA.

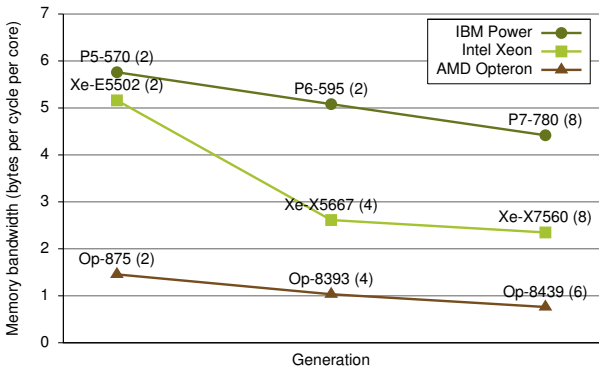

Figure 1: Theoretical memory bandwidth per core. The advent of multi-core architectures exacerbates the memory wall. We quantify the memory wall using *bytes per cycle per core*. This unit takes into consideration the absolute memory bandwidth per chip, the number of cores, and the processor frequency.

Table 1: Architecture of processors used in Figure 1. Each of these chips include integrated, on-chip memory controllers. *C*=# of cores; *T*=# of threads; /*c*=per core; *=Off-chip cache.

Processor chip	Freq GHz	C/T	L2/c MB	L3/c MB	Memory MHz	BW/c GB/s
POWER5-570	2.20	2/4	0.95	18*	DDR2-528	12.67
POWER6-595	4.20	2/4	4.00	16*	DDR2-667	21.34
POWER7-780	3.86	8/32	0.25	4	DDR3-1066	17.06
Xeon E5502	1.86	2/2	0.25	2	DDR3-800	9.60
Xeon X5667	3.06	4/8	0.25	3	DDR3-1333	8.00
Xeon X7560	2.27	8/16	0.25	3	DDR3-1333	5.33
Opteron 875	2.20	2/2	1.00	0	DDR-400	3.20
Opteron 8393SE	3.10	4/4	0.50	1.5	DDR2-800	3.20
Opteron 8439SE	2.80	6/6	0.50	1	DDR2-800	2.13

per-core and per-thread memory bandwidth will decrease over time.

More cores sharing the available memory bandwidth in and between sockets means fewer bytes per flop per core. More cores also tax a network interface controller (NIC) more. However, the resource contention is disproportional. The more cores on a node, the more communication will take place inside the node, using the memory access channels, rather than going off-node through the NIC. The amount of traffic through the NIC is related to the amount of memory per node, while the amount of traffic on the memory bus is related to the number of cores on a node. The number of cores per socket is increasing more rapidly than the amount of memory per socket.

Cache injection alleviates the memory wall by placing data from an I/O device directly into the cache [9, 17, 12]. 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. The benefits of cache injection are reduced memory access latency on I/O data and reduced memory pressure – number of requests satisfied by main memory per unit of time. This technique, however, is prone to cache pollution since the data transfer is initiated by the producer (e.g., a NIC) rather than the consumer of data. Cache injection affects performance to the degree of which I/O data is responsible for its share of the overall memory pressure.

Even though cache injection can reduce memory access latency, in this paper we concentrate more on its bandwidth benefits due to reduced memory pressure. We do this because network latency (micro seconds) dominates memory access times (nano seconds). There are certain events, however, that benefit from the improved latency, e.g., when polling a NIC for new message headers. Overall, cache injection for network I/O has a greater impact on bandwidth rather than latency.

Prefetching can hide memory latency by overlapping memory accesses with computation [6, 26]. This technique anticipates memory accesses based on usage patterns or specific instructions issued by the compiler, the OS, or the application. Unlike cache injection, prefetching 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 [12].

Chip manufacturers have expressed interest in cache injection to reduce data access latency. Recent Intel architectures provide a mechanism called Prefetch Hint [31] which allows early prefetching of I/O data initiated by the NIC. Like prefetching, however, this technique does not reduce memory bandwidth utilization. The PERCS system [5] designed by IBM allows the interconnect to write data directly into the L3 cache. This system, however, is not yet available.

In previous work, we proposed policies to leverage the benefits of cache injection by selectively injecting into the caches [12] – an injection policy selects the data to inject and the target device in the memory hierarchy. This previous work, however, neither implements/simulates the proposed policies nor evaluates their impact on performance. In the current paper, we present the evaluation of easy to implement, yet non cache-intrusive, injection policies and their effect on the performance of two applications and several collective operations. The main contributions of this paper are:

- An empirical evaluation of three injection policies on two applications and several collective operations.
- An analysis of cache injection performance in terms of the communication characteristics of applications.
- An investigation of cache injection performance on current and (expected) future systems' memory wall.
- The design and implementation of low-overhead injection policies based on the header and payload of a message.

Overall, our results shows that the effect of cache injection on application performance is highly dependent on the communication characteristics of applications, the injection policy, and the severity of the memory wall. Our characterization of cache injection can be used as a framework to determine the usefulness of this technique on parallel applications based on their communication characteristics.

In the next section, we describe our cache injection implementation and the injection policies we use throughout

the paper. Section 3 describes our test environment including a brief description of our simulation infrastructure and the applications we use. In Section 4, we evaluate key factors that determine the effectiveness of cache injection on performance. Section 5 describes the related work and we conclude in Section 6.

2. CACHE INJECTION

Cache injection [9, 17] is one of several techniques to mitigate the imbalance between processor and memory speeds [26, 27, 24]. This technique reduces memory access latency and memory pressure by placing data from the NIC directly into the cache. Cache injection, however, is prone to cache pollution since the data transfer is not initiated by the consumer of data [12]. In this section, we describe important trade-offs we face in our design of cache injection and the policies we implement on the NIC to determine when and what data to inject into the cache.

We use IBM's Power5 architecture [33], a multi-core system with an integrated, on-chip memory controller (see Figure 2), and extend it with cache injection. This architecture provides a hierarchy of caches, some local to a specific core (L1) and others shared by a number of cores (L2 and L3). We chose the L2 and L3 caches for injection because the larger cache size reduces the probability of evicting an application's working set, and the shared property reduces complexity on the NIC to determine the specific core that will consume the data. The cost associated with these choices, as opposed to choosing a local cache, is a higher access latency.

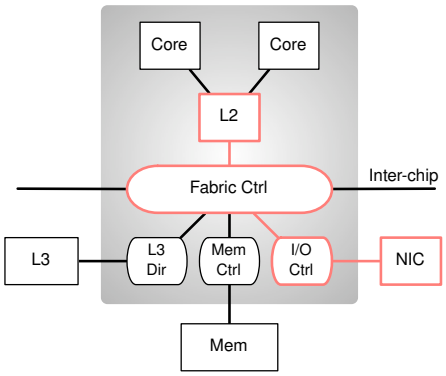

Figure 2: Base architecture.

When cache injection is enabled, data can be injected to both memory and cache, or only to the target cache. In the former, the state of the appropriate cache block is set to clean-exclusive, while in the latter, modified-exclusive. We chose to inject to both memory and cache, to reduce the number of write-backs to memory when the cache line is evicted and the data is not modified.

The base architecture provides data and control paths to the cache through the fabric controller. When moving incoming network data to the host, the NIC issues invalidation requests to the cache and data requests to main memory. When cache injection is enabled, invalidation requests are replaced with allocate/update requests. The fabric controller forwards these requests and the associated data to the cache.

Since cache injection may replace an application's working set with network data (pollute the cache), we implement a set of policies to determine the appropriate place in the memory hierarchy (L2, L3, or main memory) and discern the data to inject. These policies, implemented on the NIC, are tailored for message-passing applications.

The first policy, denoted **header** or **hl2**, writes message headers (communication events) to the L2 cache. This policy is based on the interaction between the host communication library (MPI-MIAMI [13]) 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 creates an event with the operation's parameters and signals the NIC using memory mapped registers. The NIC pulls the event from host memory, initiates the requested operation, and writes a response back 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 reduce the risk of evicting part of the application's working set from the cache, only one cache line is injected into the L2 cache per communication event.

This policy can be generalized to other communication libraries where the NIC writes small amounts of data (headers) concerning specific communication operations. Most architectures follow this model either by issuing an interrupt or by polling a queue. In our implementation, we use polling. A similar approach has been explored by writing TCP/IP headers into the cache [17, 20], but that work was 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, denoted **payload**, injects message data (payload) into the L3 cache. The performance of this policy depends on the amount of data received from the network and the timely consumption of this data by the application (message locality). Since an application's message data is not bound to a particular size, data is injected to the larger L3 cache to avoid polluting the L2 cache. The trade-off is a higher latency than the L2. This policy does not inject message payloads of more than half the size of the L3 cache to reduce pollution. Also, only full cache blocks are injected into the cache (partial blocks are written into main memory). Injecting partial blocks that are not present in the cache would require fetching the block from the lower levels of the memory hierarchy. With this design, all messages that are smaller than a cache block are written to main memory.

The third policy, denoted **hl2p**, is the union of both the payload and header policies; i.e., inject communication events or headers into the L2 cache, and inject message payloads into the L3 cache.

Section 4 shows that the performance of these policies on application performance is dependent on several application characteristics including the size and frequency of messages, the type of communication operations, and the ratio of communication to computation.

3. TEST ENVIRONMENT

Using simulation, we evaluate cache injection by executing MPI applications on a variety of system configurations and injection policies. We use a cycle-accurate framework to simulate a cluster comprised of hundreds of PowerPC computational nodes with cache injection connected through a high-performance network model [13]. Our target system configuration is shown in Table 2. The target cluster is interconnected using a fully connected network with the communication characteristics of a Cray XT-3 Red Storm machine [32]. The Red Storm represents a well-known supercomputer network.

Table 2: Target system configuration.

Feature	Configuration
Processor	POWER5 with cache injection [33, 9], 1.65 GHz frequency
L1 cache	Inst.: 64 kB/2-way; Data: 32 kB/4-way
L2 cache	1.875 MB, 3-slice, 10-way, 14 cycles latency*
L3 cache	36 MB, 3-slice, 12-way, 90 cycles latency*
Block size	128 bytes
Memory	825 MHz on-chip controller, 275 MHz DDR2 1,024 MB, 230 cycles latency*
OS	K42 [4]
Network-stack	MPICH_1.2.7-MIAMI with OS-bypass and zero-copy [13]
Network	Fully connected Cray XT-3 Red Storm

*measured latency using lmbench [25].

A detailed description and evaluation of the cluster simulator we use can be found in the literature [13]. Briefly, every node in the target cluster is simulated using two components: Mambo [8], IBM's PowerPC full-system simulator, and a simulated NIC module that attaches to Mambo. The simulated NIC implements the network model of choice (Red Storm) and the injection policies. The simulated nodes run a full-OS and communicate with each other through MPI using OS-bypass and zero-copy transfers. The cluster simulator itself runs as an MPI job on a host cluster. Note, however, that the configurations of the target and the host clusters are independent.

We run the target cluster on a host machine comprised of 4,480 compute nodes. They are dual 3.6GHz Intel EM64T processors with 6 GB of RAM. The 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. We use OpenMPI [15] version 1.2.7 and OFED version 1.3.1 to connect to the Infiniband fabric.

We use two high-performance parallel applications for our evaluation: AMG from the Sequoia acceptance suite [22] and FFT from the HPC Challenge benchmark suite [23].

AMG [16] is an algebraic, multigrid solver for linear systems arising from problems on unstructured grids. The communication and computation patterns of AMG exhibit the surface-to-volume relationship common to many parallel scientific codes. We choose AMG because it spends a significant amount of time of its execution time using the MPI library to transfer messages. This code is run 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 one. This problem size is selected due to the real-time overhead of running this application in our simulation infrastructure.

AMG has three phases of operation. The solver runs in the third phase (solve phase), while the first two phases are used for problem setup. Considering the amount of time it takes to run this application in our simulation environment, we augmented AMG to run without simulating the caches for the first two phases and enabled the caches before the third phase. Cache simulation is started early enough to allow the caches to warm-up before the solve phase. This allows us to advance the simulation quickly and to enable fully-accurate simulation just for the phase of interest.

FFT (Fast Fourier Transform) 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.

FFT measures the rate of execution of double precision, complex, one-dimensional Discrete Fourier Transform (DFT). We chose FFT because 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. We ran this code for three problem sizes: small, medium, and large; each processor computes a vector of 4,096, 65,536, and 262,144 elements respectively. Thus, the global size for a large problem on 256 nodes is 256×262,144 elements. The performance of FFT is measured in gigaflops (one-billion floating-point operations per second).

4. RESULTS AND ANALYSIS

Cache injection may benefit or hinder an application's performance. This section explores the conditions that influence application sensitivity to cache injection. We ran thousands of simulations and measured application performance in different settings, varying the factors of interest: memory and processor speed, injection policy, and node count. We used AMG and FFT as our test applications because their communication patterns differ significantly. The results in the following sections show small fluctuations between runs due to OS noise introduced by K42 [14]; e.g., different memory layouts due to demand paging. In all cases, we issued at least three runs for every experiment described in each section.

4.1 Simulation parameters and initial experiments

Over time, processor frequencies and memory access speeds have improved, although not at the same rate. For multicore processors we expect this trend to continue with the caveat that the per core bandwidth to memory will actually decrease. Physical limitations, among them pin count, will limit the memory bandwidth into a chip. As the number of cores inside the chip increases, the available bandwidth per core will go down. In Figure 1 we saw this trend of reduced memory performance in the number of bytes per cycle delivered to each core in some current and recent CPUs.

For our AMG experiments we simulated various CPU frequencies and memory speeds. We reduced the memory speed

from our baseline system (Table 2) to bring them in line with the bytes-per-cycle peak performance for each core of current and near future processors. Table 3 lists the memory and DRAM controller frequency ratios we used.

Table 3: Memory speeds used for AMG experiments. Memory frequencies are expressed as fractions of processor speed frequency.

Slowdown factor	Mem. controller frequency	DRAM frequency	Bytes per cycle per core
1.0	1/2	1/6	6.0 B/c
2.5 ×	1/5	1/15	2.4 B/c
5.0 ×	1/10	1/30	1.2 B/c
7.5 ×	1/15	1/50	0.8 B/c

Figure 3 illustrates the range of our experiments using a rectangular area around the sixteen parameter combinations we used for CPU frequency and memory speed. We also plot the CPUs listed in Figure 1 to show where their performance characteristics place them in relation to our experimental range. Our simulations span the full range of memory access speeds as represented by the nine processors from Figure 1. Our CPU frequencies are lower than several of the representative processors. The reason is that we had difficulties booting K42 inside our version of the Mambo simulator at frequencies higher than 2.1 GHz. As demonstrated below this should not invalidate our results because absolute CPU speed has very little impact on the relative effectiveness of cache injection.

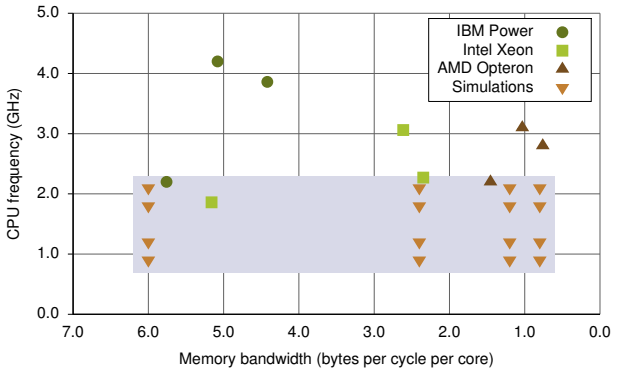

Figure 3: The shaded rectangle shows the simulation parameter range explored. Individual experiment series are marked by downward-pointing triangles. Other points in the graph mark the performance characteristics of the processors from Figure 1.

For this paper we consider four cache injection policies: base, hl2, hl2p, and payload. The base policy represents our starting point; i.e., no cache injection. We have simulated all four policies with all combinations of the memory frequency dividers from Table 3, and CPU frequencies of 900 MHz, 1,200 Mhz, 1,800 Mhz, and 2,100 Mhz. We ran each simulation at least three times to calculate a minimum, average, and maximum. Space prevents us from showing all results. Instead, we focus on three key findings for AMG.

The first question we wanted to answer is which cache injection policy works best for AMG. Figure 4 shows that hl2p is the winner. The figure shows all four policies for two node counts and memory running at full speed and slowed down to 0.8 bytes per cycle per core. Because we expect future generation multicore processors to deliver less memory bandwidth to each core, we are interested in the impact of cache injection in such systems.

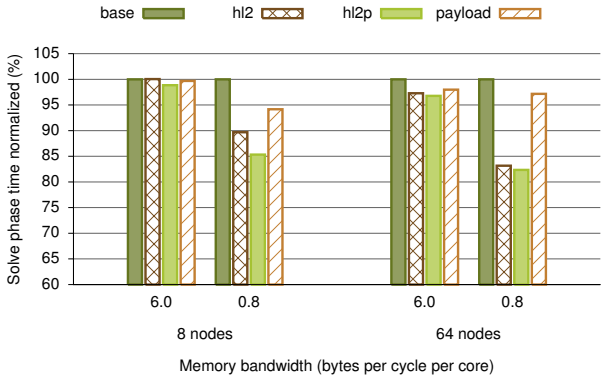

Figure 4: Performance gain of each cache injection policy for AMG for two different node counts and two memory peak bandwidths on 2,100 Mhz processors. Each bar is the minimum value of three or more runs. Memory speed has a significant impact on the base performance of AMG which is not visible in this normalized plot. This method of presentation is chosen to highlight the benefits of cache injection independent of the absolute performance of the application.

For Figure 5 we chose a 2,100 MHz processor, ran AMG on 8 and 64 nodes, and analyzed the performance when available memory bandwidth is reduced. Because we know that, for AMG and this particular workload, the hl2p policy is best we only show the base and hl2p policy simulation runs.

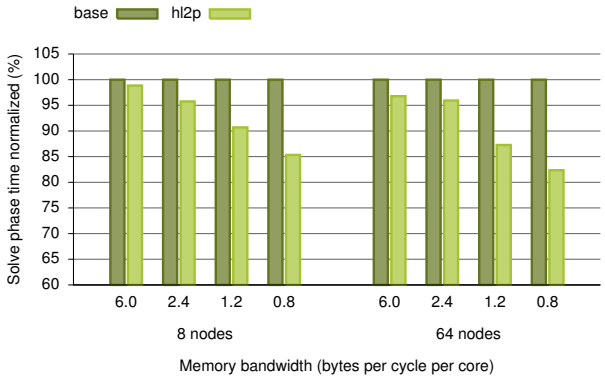

Figure 5: Impact of memory bandwidth and cache injection on AMG performance.

For both the 8 and the 64-node runs, cache injection becomes more beneficial when memory bandwidth is reduced. This particular example demonstrates a reduction in the execution time of the solve phase of AMG of more than 15%. This is a strong indication that cache injection can play an important role in future multicore systems where the memory wall for each core will be higher than it is today.

The last question to be answered in this section is whether cache injection performance depends on processor speed. In Figure 6 we present four different processor frequencies, each coupled with our standard frequency memory system from Table 2 and our worst case where we slow down the memory system frequency to 0.8 bytes per cycle per core. The figure shows that processor speed has no impact on the benefits of cache injection; it is the processor speed to memory bandwidth ratio that shows a significant gain for cache injection.

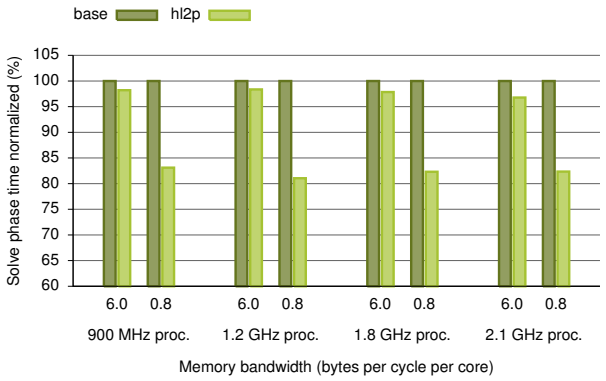

Figure 6: Impact of processor speed to memory bandwidth fraction and cache injection on AMG performance.

We conducted a series of tests on FFT to evaluate the impact of cache injection policy and show the result in Figure 7. We ran various problem sizes and show the performance results for the smallest and largest matrix we ran. The results omitted from the graph for the intermediate sizes fall between the two extremes. We ran each experiment at least three times on node sizes between 8 and 256 and show the best performance achieved. FFT reports results in gigaflops per second. To allow easier comparison to our AMG results we invert the FFT results in our graph; i.e., shorter bars indicate higher performance.

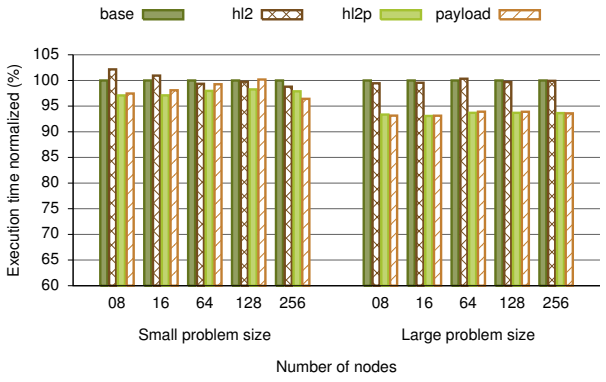

Figure 7: Performance of each cache injection policy for FFT for the small and large problem size and increasing number of nodes.

For FFT, hlp2 is still a good policy, but we make two observations. For the small problem size, the performance gains from cache injection are less clear. In some cases, for example hl2 on 8 nodes, cache injection decreases performance. When we grow the problem size, the results become more distinguishable as we can see on the right side of Figure 7. The hl2p and payload policy clearly improve performance. In contrast to AMG it is hl2p and payload instead of hl2p and hl2 that are the clear winners. In the following sections we analyze the difference in application behavior to provide an explanation for this.

4.2 Application communication characteristics

Why do the header policies (hl2 and hl2p) affect AMG and FFT differently than the payload policies (hl2p and payload)? We used the mpiP library [35] to characterize the communication behavior of our applications. Figures 8 and 9 show the computation to communication ratio for AMG and FFT. Scaling up AMG increases the time it spends communicating relative to compute time. AMG's literature [22] states that AMG can, on large number of nodes, spend more than 90% of its time communicating. FFT on the other hand, has a near constant, and fairly low, communication share of the total execution time.

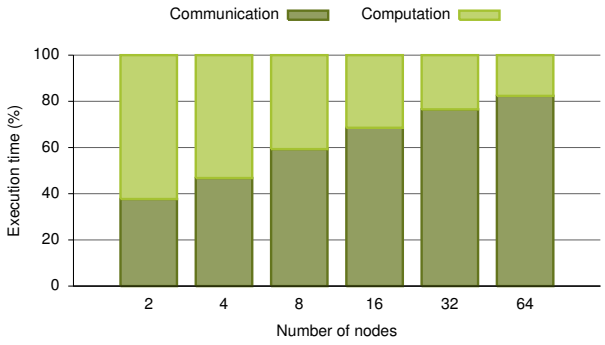

Figure 8: Communication to computation ratio for AMG.

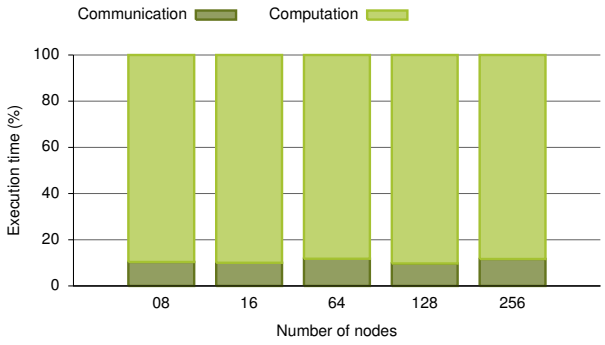

Figure 9: Communication to computation ratio for FFT.

The increasing importance of communication for AMG explains why, in Figures 4 and 5, the hl2p policy provides a bigger performance gain on 64 nodes than on 8 nodes. For FFT, the performance improvement shown in Figure 7 is nearly constant and independent of the number of nodes used; particularly for the larger problem size.

The next step in understanding why cache injection policy affects AMG and FFT differently, is to analyze the types of communication operations used by the two applications. Figure 10 shows that AMG divides its communication time about evenly among waiting, receiving, and sending point-to-point messages. Investigating the messages sizes used by AMG we learned that the relatively few `MPI_Allreduce()` operations used are all 8 bytes in length. The size of the point-to-point operations decreases as the number of nodes used grows. At 64 nodes, the average message length is 152 bytes.

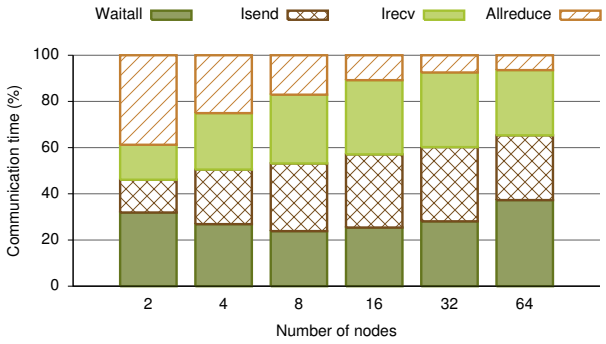

Figure 10: Types of communication operations used by AMG.

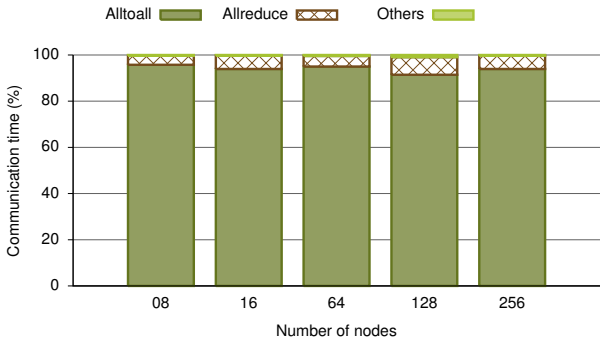

Figure 11: Types of communication operations used by FFT.

For the problem and node sizes we ran using AMG, messages are frequent and small. That explains why the header policies – hl2 and hl2p – work well for AMG. FFT's communications are dominated by `MPI_Alltoall()` (see Figure 11). The average message length of these operations decrease as the number of nodes used increase. However, even at 64 nodes, the average message size of an `MPI_Alltoall()` used by FFT is still 64 kB. That is a large enough message so that processing the payload takes much more time than the inspection of the incoming header. Therefore, FFT benefits from the payload policies. The hl2p policy combines hl2 and payload and is therefore beneficial to both AMG and FFT.

4.3 Memory reads and cache misses

When we began this research project we expected that cache injection can reduce the number of memory reads by supplying data in the cache that is used right away [13]. In this section we explore the cache and memory access behavior of AMG and FFT and relate the observed cache injection benefits to the reduced number of cache misses and memory reads.

For the large problem size runs on the right hand side of Figure 7 we show in Figures 12 and 13 the number of memory reads and cache misses. FFT issues many more reads than shown in Figure 12, but they are satisfied by the caches. The reads shown in this figure are the ones that were satisfied by main memory.

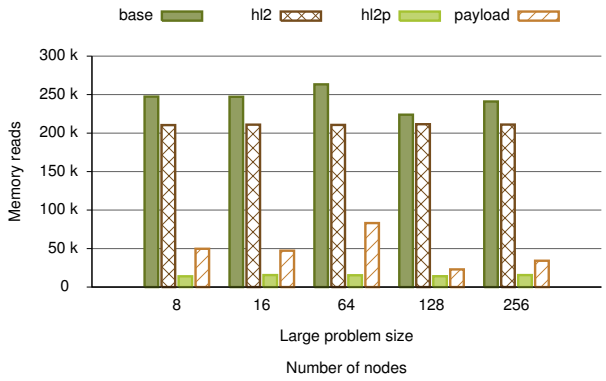

Figure 12: Number of memory reads of FFT on 1,650 Mhz processors.

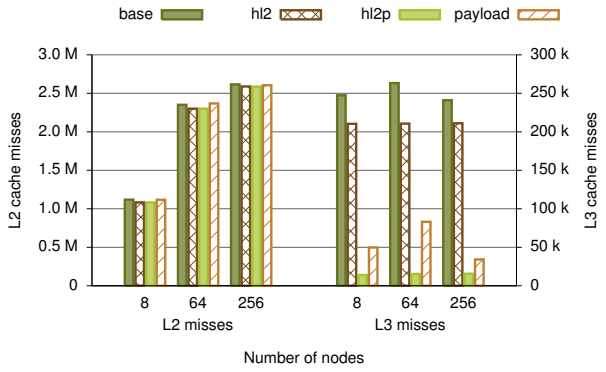

Figure 13: Number of cache misses of FFT's large problem size on 1,650 Mhz processors. The reduced misses by the header policy in the L3 cache appear much larger than the L2 due to the different scale used in the y axes of each cache.

Figure 13 shows that the injection policies are not intrusive to the L2 and L3 caches. To the contrary, the hl2 policy reduces a very small fraction of L2 and L3 misses while the payload policies reduce significantly the number of L3 misses. This behavior is the result of the relatively large message sizes used by FFT, the number of bytes injected by the payload policies is much larger than those by the header policies. Although the hl2 policy reduces a very small fraction of the L2 misses, this reduction "propagates" to the L3 cache and main memory as shown in Figure 12 (misses on network data are compulsory misses in the base case). Similarly, the reduction of L3 misses by the payload policies

results in a reduced number of memory reads. The large fraction of reduced L3 misses/memory reads explains the improved performance of FFT under the payload policies.

Figure 14 replicates the 64-node results from Figure 4 and inserts the data for the two other memory speeds we measured. This better illustrates that the header policies keep improving AMG's performance as the CPU to memory frequency ratio increases. A higher ratio means fewer bytes from memory per floating-point operation of the CPU. The payload policy is not really affected by this. To better understand the effect of the different policies on performance across the different CPU to memory ratios, we study the number of cache misses and memory reads as shown in Figures 15 and 16.

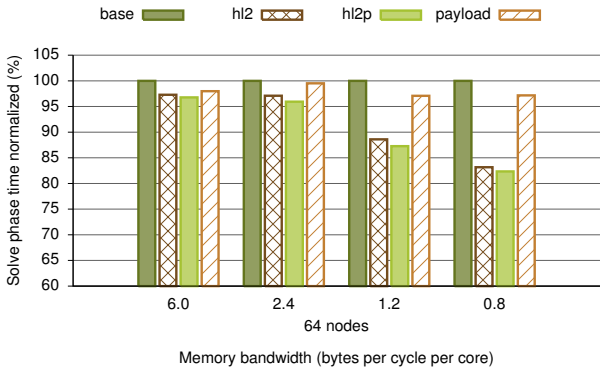

Figure 14: Performance of AMG on 64 nodes and 2,100 Mhz processors.

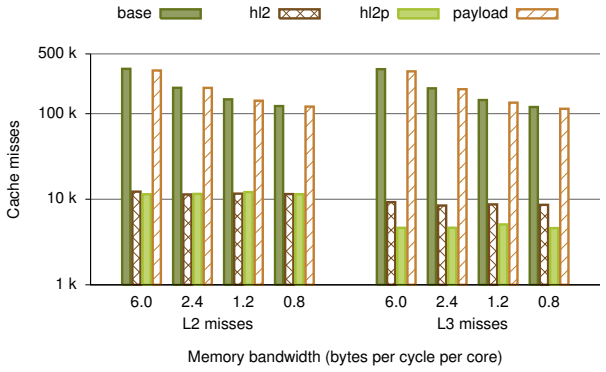

Figure 15: Number of cache misses of AMG on 64 nodes and 2,100 Mhz processors. We use a logarithmic scale on the y axis to show more clearly the behavior of the header policies.

From Figure 15 we make several observations. First, as expected, the payload policy does not affect the L2 cache. Second, the payload policy slightly reduces L3 misses as shown by the hl2p policy; this reduction is not visible from the payload policy itself due to the logarithmic scale of the y axes. A small amount of data is actually injected to the L3 cache since most messages are short and payloads of less than a cache block are written to main memory only (see Section 2). Third, the header policies largely reduce the number of misses. As shown by Figure 16 these misses are

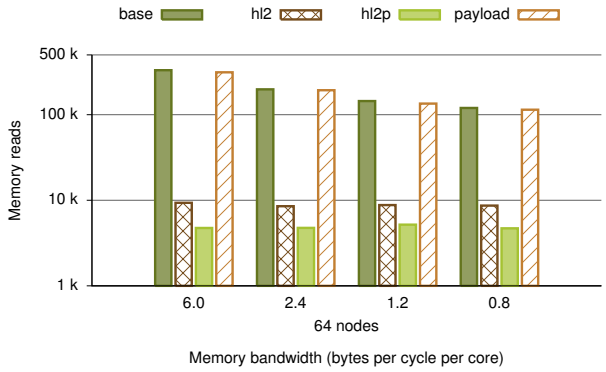

Figure 16: Number of memory reads of AMG on 2,100 Mhz processors. We use a logarithmic scale on the y axis to show more clearly the behavior of the header policies.

compulsory since they propagate from the L2 to the L3 and to memory. The large reduction of cache misses/memory reads confirms the performance improvement of AMG under the header policies.

We now focus on the behavior of the caches and memory across the different memory bandwidths studied (see Figures 15 and 16). The number of reads that the header policies cannot fulfill from cache is relatively constant across the various memory speeds. The number of cache misses and memory reads for base and payload is reduced as memory gets slower. To understand this behavior, we consult Figure 17 which shows the number of events that travel across the memory bus between the CPU and the NIC.

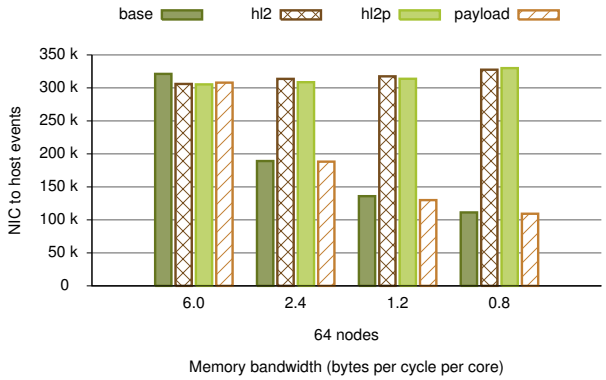

Figure 17: Number of NIC to host communication events of AMG on 2,100 MHz processors.

AMG, with our test cases, transmits a lot of messages and spends a considerable amount of time waiting for new information to arrive. The `MPI_Waitall()` function polls the NIC to learn whether a send has completed or a new message has arrived.

When we lower the available memory bandwidth the CPU can process fewer of these events because it is hitting the memory wall. This is visible for base and payload which process fewer NIC events as memory gets slower. That means the reduced number of reads across memory speeds in Figure 16 by the payload policy is simply an indication that

the MPI library polls the NIC less frequently due to memory constraints.

When header policies are used, the CPU can process an almost constant number of NIC events and is not hampered by reduced memory speed. The reason is that the CPU fetches these events from the L2 cache.

In summary, the different injection policies for the applications studied here are not cache intrusive, to the contrary, they can decrease the number of compulsory misses due to network data and thus reducing memory traffic. Choosing the right policy, based on the communication characteristics of the application, is key to improving performance.

4.4 Collective operations

Collective operations are an important aspect of many parallel applications. We analyze the impact of cache injection on these operations to provide application and system developers with information about the potential benefits of this technique on performance based on the communication characteristics of their applications. We use a micro-benchmark that measures the average execution time of seven runs of specified collective operations. We run this micro-benchmark at least five times for every experiment and report the minimum of the runs.

Cache injection can improve the performance of parallel applications that use collectives by a) reducing the execution time of a collective operation, b) reducing the latency and memory pressure of accessing data provided by a collective operation, or c) both. The benefit provided in (a) is independent of the caller's usage of the data delivered by the collective operation, while (b) completely depends on this usage. All of our injection policies have the potential of addressing (a), while only the payload policies (payload and hl2p) can address (b).

The extent to which an injection policy can address (a) is dependent on the algorithms used to implement the collective operation, e.g., binomial tree, recursive doubling, recursive halving, and ring [34]. Figure 18 shows MPICH's implementation of Broadcast on 8 nodes for 32 kB of data. For this message size and number of processors, this operation performs a Scatter followed by an Allgather (Van de Geijn algorithm [7]). Scatter is implemented using a binomial tree, while Allgather uses recursive doubling.

As this figure shows, data traverses through several nodes before it reaches its final destination. At every step of the algorithm, a subset of the nodes is involved sending and receiving messages. Each of these nodes process incoming messages selecting, storing, and forwarding payload data (an Allreduce operation may perform a reduction operation). Cache injection can improve the performance of these operations by having their operands in cache instead of main memory. The local improvements and dependencies between nodes result in a significant reduction in the overall execution time of the collective operation.

In Figure 19 we show the impact of our injection policies on the execution time of Broadcast. The payload policies reduce execution time by up to ∼30% as a function of message size. This is due to an increased number of L3 hits for larger messages. The different peaks in this figure show the switch points between algorithms used to implement this operation – binomial tree broadcast for messages less than 12 kB, Van de Geijn with recursive doubling between 12 and 512 kB, and Van de Geijn with ring for 512 kB and greater.

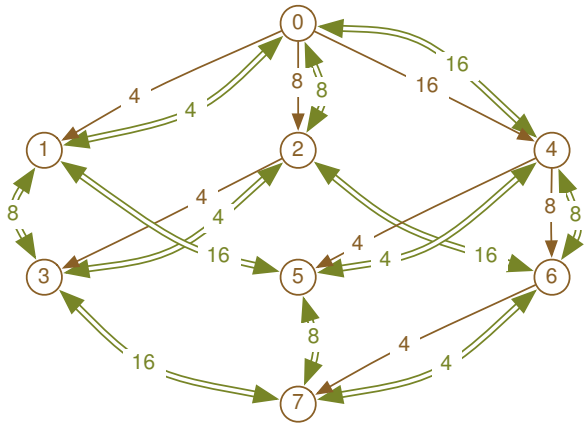

Figure 18: MPICH's broadcast algorithm for 32 kB and 8 nodes. Broadcast is implemented using Van de Geijn algorithm, which performs a Scatter (single lines) followed by an Allgather (double lines). Message sizes along the edges are given in kB. Double lines indicate communication of the specified size in both directions. Using a small number of nodes makes the algorithm easier to display.

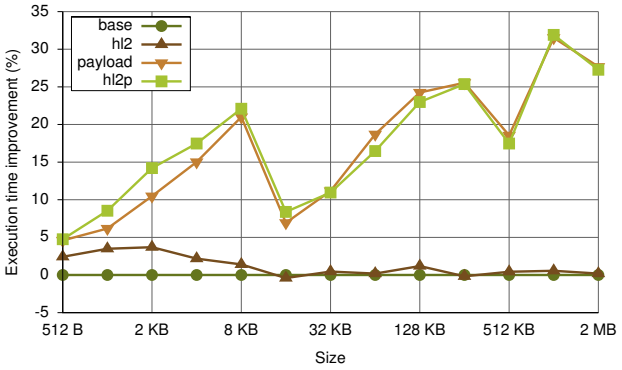

Figure 19: Effect of the different cache injection policies on the execution time of Broadcast on 64 nodes.

The hl2 policy is not particularly suited for this operation since the time to process the message's data dominates the processing of the small message header. Figure 19 shows that the improvement by the hl2 policy diminishes as the message size increases. The hl2 policy is useful in latency-sensitive operations where a significant amount of the processing per message is spent on processing the message's header. Applications using a large number of short messages meet this criteria. This is the reason hl2 improves AMG performance.

We also measured the effect of the different policies on other collective operations. We observed that hl2 has no effect on Allreduce, Allgather, Alltoall, Broadcast, and Scatter for 8, 16, and 64 nodes. We also observed that hl2p performs almost equally as payload (hl2 provides almost no improvement), and that the payload policy shows a similar pattern across node counts. Thus, in the following experiments we only describe the payload policy on the larger node count.

Figure 20 shows the impact of the payload policy on collective operations for 64 nodes. The different peaks of Broadcast, Allgather, and Allreduce show the switch points between algorithms used to implement these operations – recursive doubling, ring, recursive halving, and binomial tree. A common characteristic of these algorithms is the intermediate processing of data as messages propagate to their final destination. As we showed for Broadcast, the payload policy can improve the performance of these intermediate operations resulting in an improvement of the overall execution time. The best performing algorithms are those used by Broadcast. The graph also shows, implicitly, the improvement of Scatter, demonstrated by the improvement of Broadcast over Allgather for messages of at least 12 kB – Broadcast uses the Van de Geijn algorithm, which performs a Scatter followed by an Allgather.

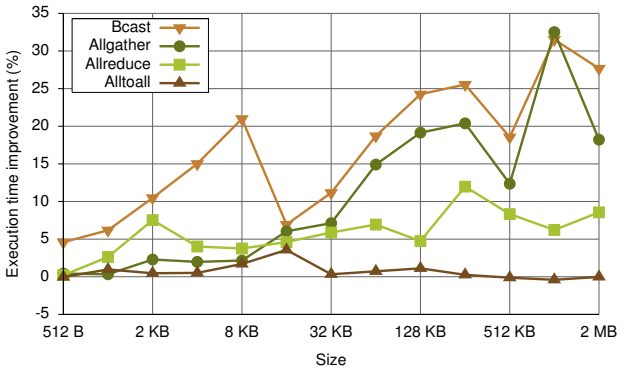

Figure 20: Performance improvement of certain collective operations using the payload policy on 64 nodes.

Figure 20 also shows that cache injection does not improve the performance of Alltoall operations. Unlike the collectives discussed above, Alltoall is implemented by a different type of algorithm [34]. For small messages of at most 32 kB, every node posts p non-blocking receives and p non-blocking sends followed by a Waitall operation, where p is the number of nodes. The data is transferred directly from the source to the final destination in one step, thus there is no intermediate processing of the data that can be exploited by the payload policy. For messages larger than 32 kB, a pairwise-exchange algorithm is used. Similarly to the small message case, there is no intermediate usage of the data during the operation.

Thus, the benefits of the payload policy on Alltoall are not realized during its execution, but can be leveraged when the operation completes. If an application uses the Alltoall data promptly after completion of the operation, the data will still be in the cache and thus, improve the subsequent computations on this data. Since this operation delivers data to all the nodes in the communicator, every node would see an improvement. This is the reason the payload policy improves FFT performance although it does not reduce Alltoall's execution time.

In summary, the payload policy reduces the execution time of Allgather, Allreduce, Broadcast, and Scatter by improving the intermediate processing of the underlying algorithms. Cache injection may improve the performance of applications strongly affected by these operations or well-known collective algorithms such as binomial tree, recursive doubling, and ring. The impact of the payload policy on applications driven by Alltoall operations is dependent on their data usage patterns.

The hl2 policy is best suited for applications exchanging a large number of small messages. These applications tend to be latency-sensitive. In addition, we believe that hl2 will improve the performance of algorithms such as the Bruck algorithm [10] used in MPICH for very small messages.

Since the impact of cache injection on the different collectives is dependent on the algorithms used to implement these operations, cache injection performance may differ among MPI implementations.

5. RELATED WORK

Bohrer et al. [9] propose cache injection as a mechanism to reduce memory access latency in processing I/O data. Using simulation, the authors show improvements in execution time for network and disk micro-benchmarks. Huggahalli et al. [17] show that cache injection of incoming network packets, termed Direct Cache Access (DCA), can reduce data access latency and memory bandwidth usage for several benchmarks including SPECweb99, TPC-W and TPC-C. Their simulation work reduced significantly the overhead of TCP/IP protocol processing.

The processing of TCP/IP packets is a desirable target for cache injection since network processing occurs immediately after writing the packet to the cache (signaled by an interrupt to the processor). DCA improves the performance of memory copies executed by the network stack. Our work focuses on high-performance computing environments where message data is usually delivered directly to the application without copies or interrupts. In this context, the application's data usage patterns determine the effectiveness of cache injection.

Kumar et al. [20] provide an initial evaluation of DCA on a real machine based on Intel's I/O Acceleration Technology (I/OAT) [31]. This technology, through prefetch hint, allows the NIC to signal the host's prefetch engine to start fetching incoming network data into the cache. Their study shows that prefetch hint can reduce processor utilization for certain micro-benchmarks. Their target environment is a data center where many applications may be running at once on the same node. Therefore, applying DCA to the processing of network packets may increase processor availability for other applications. This study has been extended to address the usage of DCA in multi-core architectures and a standard Linux network stack [21]. Unlike cache injection, the prefetch hint implementation of DCA does not reduce memory bandwidth usage.

León et al. [12] compare cache injection with data prefetching. Using simulation, the authors show that cache injection outperforms prefetching in reducing memory bandwidth utilization. This reduction is important for applications limited by memory bandwidth. The authors also propose using appropriate injection policies based on OS, compiler, and application information to minimize the amount of pollution introduced by cache injection in an HPC environment. The proposed policies, however, were not evaluated.

Khunjush et al. [19] propose the creation of a network cache to store incoming network data of small size (one cache block). Using simulation, the authors show their system can avoid memory copies due to network stack processing.

The authors gathered traces from a real system and replay them into one simulated node which implements their architectural modifications. Unlike our work, this approach is applicable only to small packets and has the cost of implementing and searching an additional cache in the system.

6. SUMMARY AND CONCLUSIONS

In this paper, we present an empirical evaluation of three types of cache injection policies – header, payload, and the combination of the two – and their effect on the performance of two applications and several collective operations.

The choice of injection policy is an important factor in the performance of cache injection. Our results show that the header policies improve the performance of AMG, while the payload policies improve the performance of FFT. The different effect on performance is directly related to the application's communication characteristics. The header policies are best suited for applications using a large number of small messages, while the payload policies are most effective on applications exchanging medium and large messages.

Our results also show that for the applications studied here, cache injection does not have a negative impact on the caches. As expected, the policies that improve performance significantly reduce the number of compulsory cache misses relative to the total number of main memory accesses. Those policies that do not improve performance keep the number of L2 and L3 misses close to the base case. Clearly, the impact on the caches is application dependent – an application needs to consume the injected data promptly. Many applications, however, have this property. For example, codes using collective operations will likely use the network data shortly after the operation completes.

To help identify types of applications that may benefit from cache injection based on their communication characteristics, we investigated the impact of this technique on collective operations. The payload policies reduce the execution time of Broadcast, Allgather, Allreduce, and Scatter by up to 30% as a function of message size. The improvement is dependent on the algorithms used to implement these operations: binomial tree, recursive halving, recursive doubling, and ring. Cache injection, however, does not improve the execution time of Alltoall operations due to the pairwise-exchange algorithm used. Despite this lack of improvement, applications using Alltoall operations can still benefit from cache injection by using the data delivered by the operation promptly after completion. This is the reason the payload policy improves FFT performance although it does not reduce Alltoall's execution time.

Thus, based on the communication characteristics of applications, one can make an informed decision about the potential benefits of this technique on performance. This provides a metric for application developers to determine whether to enable/disable cache injection when it becomes available. Furthermore, our work shows that cache injection hardware will be more broadly useful and perform better if it supports custom replacement policies for specific applications and different programming models, rather than just the ability to turn it on or off.

Since we expect future generation multicore processors to deliver less memory bandwidth per core, we also explored the impact of cache injection on systems with different processor and memory speed configurations. Our results indicate that cache injection can play an important role in these systems where the memory wall for each core will be higher than it is today. Furthermore, understanding the trade-offs of this technique is important as new architectures capable of performing cache injection, such as the PERCS high-performance interconnect, become available.

For future work, we are interested in analyzing a wider range of applications and problem sizes. We also plan on investigating dynamic policy reconfigurations. Using "hints" from applications and the MPI library, the NIC may determine automatically, for every message, the target device in the memory hierarchy. A simple example, a generalization of the policies studied here, is a dynamic policy that selects the target cache based on the size of incoming messages. A more complex example involves using hardware counters and other information so that the NIC can dynamically adapt when the caches are polluted.

7. REFERENCES

[1] AMD Inc. Opteron 875, Opteron 8393SE, and Opteron 8439SE. `http://www.amd.com/`, Mar. 2010.

[2] G. Anselmi, B. Blanchard, Y. Cho, C. Hales, and M. Quezada. IBM Power 770 and 780 technical overview and introduction. Technical Report REDP-4639-00, IBM Corp., Mar. 2010.

[3] G. Anselmi, G. Linzmeier, W. Seiwald, P. Vandamme, and S. Vetter. IBM system p5 570 technical overview and introduction. Technical Report REDP-9117-01, IBM Corp., Sept. 2006.

[4] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source Linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[5] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS high-performance interconnect. In *Symposium on High-Performance Interconnects (Hot Interconnects)*, Aug. 2010.

[6] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *ACM/IEEE conference on Supercomputing (SC)*, pages 176–186, Albuquerque, New Mexico, 1991.

[7] M. Barnett, L. Shuler, S. Gupta, D. G. Payne, R. A. van de Geijn, and J. Watts. Building a high-performance collective communication library. In *Supercomputing*, pages 107–116, 1994.

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

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

[10] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Symposium on*

Parallel Algorithms and Architectures (SPAA), pages 298–309, 1994.

[11] C. Cler and C. Costantini. IBM Power 595 technical overview and introduction. Technical Report REDP-4440-00, IBM Corp., Aug. 2008.

[12] E. A. León, K. B. Ferreira, and A. B. Maccabe. Reducing the impact of the memory wall for I/O using cache injection. In *Symposium on High-Performance Interconnects (Hot Interconnects)*, Palo Alto, CA, Aug. 2007.

[13] E. A. León, R. Riesen, A. B. Maccabe, and P. G. Bridges. Instruction-level simulation of a cluster at scale. In *International Conference on High-Performance Computing, Networking, Storage and Analysis (SC)*, Portland, OR, Nov. 2009.

[14] K. B. Ferreira, R. Brightwell, and P. G. Bridges. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *2008 ACM/IEEE Conference on Supercomputing (SC)*, November 2008.

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

[16] V. E. Henson and U. M. Yang. Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2000.

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

[18] Intel Corp. Xeon E5502, Xeon X5667, and Xeon X7560. `http://ark.intel.com/`, Mar. 2010.

[19] F. Khunjush and N. 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, Aug. 2007.

[20] A. Kumar and R. 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, Dec. 2007. IEEE Computer Society.

[21] A. Kumar, R. Huggahalli, and S. Makineni. Characterization of direct cache access on multi-core systems and 10GbE. In *15th International Symposium on High-Performance Computer Architecture (HPCA'09)*, Raleigh, NC, Feb. 2009.

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

[23] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite, Mar. 2005.

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

[25] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, Jan. 1996.

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

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

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

[29] R. C. Murphy and P. 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.

[30] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware. In *USENIX Annual Technical Conference*, pages 247–256, 1990.

[31] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, Nov. 2004.

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

[33] 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.

[34] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[35] J. S. Vetter and M. 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.

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