Optimizing a Multi-Core Processor for Message-Passing Workloads [*]

Niladrish Chatterjee, Seth H. Pugsley, Josef Spjut, Rajeev Balasubramonian
School of Computing, University of Utah
{nil, pugsley, sjosef, rajeev}@cs.utah.edu

Abstract

Future large-scale multi-cores will likely be best suited for use within high-performance computing (HPC) domains. A large fraction of HPC workloads employ the message-passing interface (MPI), yet multi-cores continue to be optimized for shared-memory workloads. In this position paper, we put forth the design of a unique chip that is optimized for MPI workloads. It introduces specialized hardware to optimize the transfer of messages between cores. It eliminates most aspects of on-chip cache coherence to not only reduce complexity and power, but also improve shared-memory producer-consumer behavior and the efficiency of buffer copies used during message transfers. We also consider two optimizations (caching of read-only and private blocks) that alleviate the negative performance effects of a coherence-free system.

Keywords: *MPI, cache coherence, private and read-only blocks, buffered and bufferless copy for MPI.*

1. Introduction

High-performance processors of the future will inevitably incorporate hundreds of processing cores. Already, there are plans for processors with 64 cores (Tilera [26]), Intel's 80-core Polaris prototype [27] and even graphics engines with 960 cores (NVIDIA Tesla S1070 [24]). Most computer scientists agree that the biggest roadblock to the widespread adoption of large-scale multi-cores is a lack of multi-threaded software [4]. There are concerns on multiple fronts: (i) What kinds of desktop/laptop applications are amenable to parallelization across hundreds of cores? (ii) How do we develop programming models that make it easier for programmers to partition applications across hundreds of threads? If there are no satisfactory answers to these questions, most desktop/laptop processors will accommodate tens of cores and large-scale multicores with hundreds of cores will likely be used exclusively in the high-performance computing (HPC) and server domains.

In the HPC domain, several applications with high degrees of thread parallelism have already been developed and highly tuned. These (mostly legacy) applications have been developed with the message-passing interface (MPI) so they can be executed on hundreds of CPUs across a cluster of workstations. It is extremely likely that when hundred-core CPUs are made available, they will be mostly employed to execute these legacy MPI applications – not only are these applications highly parallel, the code-base for them already exists. It is therefore important that the architecture of large-scale multi-cores be significantly revamped to target the important class of MPI workloads.

If current trends continue, the architecture of future multi-cores will very likely resemble the following. Each core will have private L1 instruction and data caches. Many cores will share a large L2 cache that may be physically distributed on chip, allowing each core to have quick access to a portion of the L2 cache [20, 28]. Intelligent page coloring will ensure that L1 misses are serviced by a portion of L2 that is relatively close by [3, 13]. Cache coherence will be maintained among the single L2 copy of a block and possibly numerous L1 cached copies. To allow scalability across hundreds of cores, a directory-based coherence protocol will be employed [14]. Each L2 block will maintain a directory to keep track of the L1 caches that have a copy of the block.

Today, there is an over-riding sentiment that every multiprocessor must support cache coherence in order to be able to execute shared-memory programs. The OS is one such shared-memory program that executes on every system. However, there is a fairly hefty price being paid to implement cache coherence [14]: (i) storage overhead for the directory, (ii) controller logic to implement the non-trivial operations associated with each coherence message, (iii) the delay/power overheads and indirection involved in accessing blocks via a directory-based protocol. The value of cache coherence is highly diminished in a processor that almost exclusively executes MPI workloads. Hence, there is a need for an architecture that does not incur the usual overheads of cache coherence when executing MPI workloads, but that ensures correctness when occasionally handling shared-memory code (the OS, the MPI runtime, etc.).

MPI applications and run-times typically assume generic off-the-shelf hardware to allow parallelization across an arbitrary cluster of workstations. This model will change in the future if the entire application executes on one (or a few) aggressive multi-core. The run-time libraries need not rely on generic inter-processor communication primitives, but can leverage special primitives hardcoded within the multi-core. The plentiful transistor budgets in future multi-cores allow for such hardware specialization. The design of efficient inter-processor communication primitives would therefore be an important component in designing a processor optimized for MPI workloads.

[*] This work was supported in parts by NSF grants CCF-0430063, CCF-0811249, NSF CAREER award CCF-0545959, Intel, SRC grant 1847.001, and the University of Utah.

In this position paper, we present a chip that is optimized for the execution of MPI workloads. It not only removes the negative impact of cache coherence on the execution of MPI programs, it also adds support for efficient messaging between MPI threads. Efficient MPI messaging on a multi-core is typically hampered by cache coherence; the elimination of cache coherence therefore aids in the design of efficient messaging mechanisms. Thus, the two innovations in this paper are synergistic.

This position paper primarily puts forth the proposed architecture with almost no quantification of the expected benefits. In places, we do provide some quantification as evidence of promise. The paper is organized as follows. Section 2 discusses related work. Section 3 describes how the overheads of cache coherence can be largely eliminated. Section 4 describes the hardware support to accelerate message-passing between threads. We draw preliminary conclusions in Section 5.

2. Related Work

To date, very few multi-processor systems have considered eliminating cache coherence. In small-scale multi-processor systems, shared-memory programs are expected to be prevalent and the overheads of cache coherence are considered worthwhile. Exceptions include IBM's Cell [15] that relies on software to manage memory transfers to and from local memory. The Cell avoids dealing with cache coherence by treating local store memory as separate memory locations. Intel's 80-core Polaris prototype [27] was designed to have limited functionality and did not implement cache coherence. MPI systems have traditionally executed on clusters of workstations without any support for hardware cache coherence. The execution of MPI applications on multi-cores is a relatively recent phenomenon and these multi-cores have so far been designed to efficiently handle shared-memory applications. If future multi-cores are going to primarily execute MPI applications, it is important to architect a design that removes the overheads of cache coherence while still correctly and efficiently handling shared-memory codes (the OS and MPI run-time). This paper is the first to propose the elimination of L1 cache coherence, while still allowing some "safe" caching within L1 to alleviate occasional loss in performance.

In the rest of this section, we briefly present innovations for speeding up the communication phases of MPI programs over traditional, distributed memory clusters and also on shared memory systems.

2.1 Internode Communication

Internode communication has been accelerated by the usage of gigabit networks (Infiniband/Myrinet) and techniques like RDMA [19] (remote direct memory access) where the network card directly copies data from the sender's address space to that of the receiver, bypassing the OS. The zero-copy principle that forms the basis of RDMA is similar in its theme to our proposed innovations.

2.2 Intra-CMP Communication

The techniques for faster intra-CMP communication that have been proposed in existing literature can be classified as: (i) NIC-based loopback, (ii) User space memory copy and (iii) Kernel assisted memory copy

Of these, the NIC-based technique is the most naive technique, where the NIC on detecting that the destination of the message is on the same node as the sender, simply loops the message back instead of injecting the message into the network . As the message has to pass through the entire TCP/IP protocol stack, it has high latency [6] and is agnostic of the memory organization.

We next discuss the various forms of the second technique and also a case of kernel assisted copy.

2.2.1 User Space Memory copy

Baseline Dual Copy Mechanism This is the naive approach used for MPI send/receive on a shared memory architecture [11]. This involves the creation of a buffer area in the address space shared by the receiver and sender. A message is passed by copying it out of the sender's data structure into the shared buffer and subsequently into the receiver's data area. Besides the overheads of O(P^2) memory requirement (P = number of processes) and polling for data, this method suffers from cache pollution effects of multiple copies.

Enhanced Buffer Management Under this message-size guided scheme [11], small messages are transfered in the exact same way as in the previous case. But large message transfers take place in a fragmented (possibly pipelined) manner using smaller send buffer cells. The sender notifies the receiver when data is available thus eliminating the need for polling. The use of small receive buffers and the potential reuse of the small send buffers leads to better cache behavior.

The Nemesis-communication subsystem In this system [7, 8], there is a single receive buffer per process which is accessed via lock-free queuing primitives leading to smaller memory footprints and faster queuing of message segments. Intelligent organization of the shared data structures (like placing the head and tail pointers of the queue in the same cache line) as well as innovations like shadow head pointers and non-temporal stores, enable Nemesis to have the lowest latency and significantly better cache usage.

2.2.2 Kernel Assisted Message Transfer

This is a single copy mechanism [17, 16], where the OS maps the sender or receiver's private area into its own address space and performs the copy directly. The benefit of a single copy and bufferless transfer is obtained at the cost of expensive system calls. A hybrid technique comprising the usual dual-copy transfer for small messages and kernel assisted techniques for message sizes above a threshold has been proposed in [12].

3. Eliminating Cache Coherence

3.1 Motivation

As described in Section 1, cache coherence keeps several private data caches up to date with one another. Cache coherent systems incur various overheads. There is a die area cost in the form of increased storage for the directory, which scales linearly with the number of cores. This has been previously recognized as a problem and commercial multiprocessor systems have handled it by tracking information at a coarser granularity [14, 18]. That, in turn, results in broadcasts to groups of processors and increased network traffic/power. Even with these optimizations, the directory in systems with hundreds of cores can be expected to occupy several megabytes. Additionally, a fair amount of network traffic is invoked on any coherence operation, in particular any write-request can require an invalidate message to every node in the network in the worst case. In addition to the power cost, there is a significant latency penalty as well. This is especially true when cache coherence introduces an extra level of indirection to access shared data, as is described below.

3.1.1 Producer-Consumer Data Sharing

A workload that shows off the problems of cache coherence is one with producer-consumer behavior. In all producer-consumer applications, one thread populates a memory location with useful data, and another thread reads that memory location. In cache coherence terms, one core holds a block in modified state, and then another core requests to be added as a reader of that block. This request is sent to the block's directory (part of the L2 cache in a multi-core processor), which forwards the request to the core that holds the block in modified state. The producer core then forwards the block to the consumer. This triangle of coherence traffic happens every time one core attempts to consume data produced by another core. Additionally, if the same data structure is reused by the producer, then it must again get the blocks holding that data structure in modified state, which generates invalidate coherence messages. The producer cannot proceed with its write unless it receives acknowledgements from all cores that have a cached copy of that block. A directory protocol thus introduces three network messages in the critical path of many reads and writes and is therefore expensive in terms of delay and power.

3.1.2 MPI

Most MPI programs do not employ shared-memory. But if the application executes on a platform that offers shared-memory, the MPI run-time library will typically adopt the shared-memory programming model to reduce the number of intermediate copies during message transfer. The MPI run-time library's shared-memory implementation of message transfer is an important example of producer-consumer type sharing behavior. Messages are sent using buffers where the producer thread's data structure is copied into the run-time library's shared-memory buffer space. The consumer thread then copies this data from the buffers into its own local data structures. The run-time library's shared-memory buffer space is therefore repeatedly written into and then read from. Cached copies of these buffers are therefore subject to a large amount of coherence traffic. Once the buffer has been read by the consumer, the same region of memory can be reused for subsequent messages, resulting in many invalidates, especially if there are many sharers (depends on the type of MPI message).

3.2 Proposed Design – Read-Only Blocks

Cache coherence is needed because of the possibility of multiple L1 caches having copies of a cache line that has the potential of changing. There is no need for cache coherence operations if each writable block is only stored in one location that all potential readers/writers can easily access. Many benchmarks have small sets of cache accesses to lines that are both written and shared between cores, and some have very large numbers of cache accesses to read-only lines [5]. We therefore impose the restriction that each core's L1 cache store blocks that are read-only. If a block can be written into, it is placed only in the shared L2 cache and not in any L1 cache, eliminating multiple copies of this block and any coherence activity on a write or read. Accesses to read-only blocks are handled very similarly to the way they are handled in the traditional cache coherent system (without the overheads of cache coherence). Writable blocks are never found in L1 and always incur the higher latency cost of accessing L2. Before we describe the mechanisms required to identify blocks as read-only, let us consider the impact of such an organization on performance.

Since there is no change to how read-only blocks are accessed, they will frequently incur short L1 access latencies on a read, just as in the cache coherent baseline. Note that the L1 instruction cache only contains read-only blocks and the proposed model has no impact at all on the L1 instruction cache's design. Since the L1 data cache now no longer accommodates writable blocks, it can accommodate more read-only blocks than the baseline and yield a higher hit rate for such blocks. Access to writable blocks will first require a futile L1 look-up, followed by an L2 cache access. Compared to the baseline cache coherent system, this may or may not incur longer latencies. If the baseline system would have yielded an L1 hit for the block, the proposed model ends up being less efficient. If the baseline system would have yielded an L1 miss and would have required coherence operations to locate a clean cache copy, the proposed model ends up being more efficient.

It is worth noting that in future multi-cores, an L2 access may not be significantly more expensive than an L1 access. Large multi-core L2 caches are expected to be partitioned into megabyte-sized banks [22], each bank possibly associated with a core. Assuming that the OS can implement good page coloring policies, a core should be able to find its data in its own L2 bank or in a neighbor's L2 bank. L2 access time is therefore equal to the cost of a single bank's access with an occasional additional network hop or two.

According to CACTI 6.0 [23], a 64 KB L1 cache incurs a 3-cycle delay penalty (3.0 GHz clock speed at 65 nm technology) and a 1 MB L2 cache bank incurs a 6-cycle delay penalty. Thus, the elimination of L1 caching for writable blocks can have a negative effect on performance if the baseline system had yielded several L1 hits for such blocks. Our own Simplescalar [9] out-of-order processor simulations for many single-threaded SPEC programs show that each additional cycle in L1 latency for every access results in a 3% performance slowdown. The proposed model therefore has an 18% upper bound for performance degradation when using out-of-order cores (assuming all blocks are writable, there is no saving in cache coherence traffic, and page coloring allows perfect localization).

We are also presently assuming that an L1 miss must be flagged before the L2 can be accessed. This adds three more cycles to the access time of writable blocks. We believe that this negative effect can be alleviated by having a simple structure that predicts if the address is writable or read-only. If the address is predicted to be writable, the L2 access can be initiated in parallel with the L1 access.

Next, we examine how a block is categorized as read-only or writable. We adopt a simple mechanism to identify read-only blocks that relies on a two-bit counter and two additional bits for each cache line, one for denoting if the line is cached in L1, and another for denoting whether or not the line is dirty (note that the latter is already part of most caches). When a line is brought in to the cache, the counter is set to a value of two. Every N cycles, counters for all cache lines are decremented (in a saturating manner). If a line's counter has reached zero and its dirty bit is not set, we deem the block as read-only (since there have been no writes to this block in at least N cycles, where N is sufficiently large). On subsequent accesses to this block, the block may be cached in L1s and a "cached" bit associated with that line is set. Until the counter reaches zero, all accesses of that cache line are handled by the L2. If a line is evicted and brought into cache again, it has to go through this process again to determine if it is read-only or writable.

This speculative categorization of blocks can occasionally lead to mispredictions. All writes are directly sent to the L2 cache (note that this does not lead to increased L2 activity because most multi-cores are expected to adopt write-through L1 caches). If the L2 notices a write to a block that has its "cached" bit set, it detects a misprediction. At this point, a broadcast invalidate is sent to all cores so they evict that line from their caches. We are attempting to remove directory coherence overheads and therefore do not attempt to track a list of sharers. A broadcast operation that is invoked on an occasional mispredict is likely to incur minor overheads. All future accesses to that line are handled by the L2.

Our proposed architecture incurs modest overhead in terms of die area. We require 4 bits of storage per cache line in L2, instead of the many bits needed to maintain a coherence directory for each cache line. While cache coherence directories scale in size as the number of cores (and thus potential sharers) increases in the system, our 4 bit overhead is fixed and does not increase as the number of cores scales.

3.3 Proposed Design – Private Blocks

In the innovation described above, we allow L1 caching only for read-only blocks that are known to not require cache coherence operations. Similarly, there is another important class of data blocks that does not require cache coherence operations: private blocks that are read/written by only one core. Private blocks are obviously abundant in single-threaded and MPI applications. They are also extremely common in shared-memory programs that are amenable to data parallelization. By allowing private blocks to also be cached in L1, the proposed design has an even lower negative impact on performance while continuing to enjoy the benefits of a coherence-free system.

The process for the detection of a private block is very similar to that for read-only block detection. The first core to access an L2 block is recorded along with the block (an overhead of $log(num-cores)$). On every subsequent access, the requesting core id is compared against the id of the first accessor (similar to the tag comparison that is already being performed). If they ever differ, a "not-private" bit is set. If at least N cycles have elapsed and the block is flagged as private, it can be cached in L1. If a request is ever received from a different core and the "cached" bit is set, an invalidation is sent to the first accessor and the not-private bit is set.

3.4 Preliminary Analysis

The innovations in this section attempt to design a system that is free of the implementation overheads of cache coherence. Our hope is that this leads to improved performance in many workloads, and tolerable performance loss in a few workloads. The high-level impact on the performance of different workloads is summarized below:

- Shared-memory programs that have high L1 hit rates (in the baseline cache-coherent system) for writeable and non-private data will perform poorly in the new model. Since the proposed design is targeted at MPI workloads, the only shared-memory codes on the system should be the OS and the MPI run-time library and the overall impact on MPI applications may be low (we have yet to verify this).

- Other shared-memory programs that have a high degree of data parallelization should perform similarly in the proposed and baseline models as most data will be categorized as private and cache coherence operations will not be a major bottleneck.

- Shared-memory programs that have a high degree of producer-consumer sharing should perform better in the new system since the latency overheads of cache coherence are reduced (assuming that the writable blocks do not have a high L1 hit rate in the baseline system). The MPI run-time library that primarily performs buffer copies falls in this category.

- Non-shared-memory programs should behave similarly in the baseline and proposed systems. This includes single-threaded applications and the individual threads of an MPI application.

Overall, an MPI application should perform better in the proposed system because of faster buffer copies. This benefit is in addition to the lower design complexity for the processor.

We are yet to interface MPI applications with our architectural simulators. We have also not faithfully simulated our proposed design changes. In this initial analysis, we present some preliminary numbers that estimate the impact of the proposed changes on the performance of shared-memory codes.

We use SIMICS [21] with the g-cache module and run shared-memory benchmarks from the NAS Parallel Benchmarks suite. Our baseline and proposed systems are 16-core in-order systems. Note that in-order processors are much more sensitive to cache access latencies than out-of-order processors, but lead to reasonably short simulation times. The cache hierarchy of the baseline system employs a 16 MB L2 cache, and 64 KB each for private instruction and data L1 caches. Our proposed system uses the same 16 MB L2 cache, but only uses a single 64KB private instruction cache. The 16 MB L2 caches are composed of 16 1 MB banks, with each bank having 6 cycle latency, connected by a grid network. L1 accesses are assumed to have a 3-cycle latency and L2 accesses are assumed to have a 7-cycle latency (assuming that page coloring cannot localize every request to the nearest L2 cache bank). To simulate our proposed system, we first model a design where all data accesses take the full 7 cycles of an L2 access, because we do not model any read-only local cache. In a second configuration, we allow read requests to have a lower 4 cycle L2 access time, while writes continue to be 7-cycle accesses. This is obviously not a faithful representation of the read-only and private block optimizations in this paper, but shows the performance change possible if a subset of accesses (especially reads) are serviced faster.

Running the NAS Parallel Benchmarks on our baseline and first proposed configuration shows a performance decrease of between 33% and 40%, depending on the benchmark. Our second configuration has much better performance than the first, but still yields a performance degradation of 2.5% to 19.5%, relative to the baseline (note again that these simulations are for in-order cores). This indicates the importance of introducing schemes to encourage selective L1 caching such as the read-only and private block optimizations introduced in this paper. As explained earlier, shared-memory programs can expect to see a (moderate) performance loss unless they exhibit a high degree of producer-consumer sharing and that is exactly what we see in the NAS parallel benchmarks. The hope is that such shared-memory codes will not be prevalent in a multi-core designed for MPI workloads.

4. Efficient Messaging

The message passing model has been the natural choice as an IPC mechanism for large clusters over the years with MPI being its de-facto standard. While researchers have focused on reducing network latencies and improving algorithms for effective bandwidth utilization for overall speedup, there has not been much work on adapting MPI primitives to a shared memory environment. However the widespread incorporation of multicore processors as processing units in clusters, requires a rethinking of the ways in which MPI can be made to perform efficiently in such systems.

A hybrid approach of using MPI and threaded-shared-memory programming paradigms like OpenMP/Pthreads has been advocated for programming clusters of shared-memory nodes [25]. However, this would require rewriting a major volume of existing MPI applications - many of which have been carefully hand-tuned for performance over years and moreover their sheer complexity prohibits any major change of this legacy code-base.

4.1 Motivation

Multicore processors are now ubiquitous. Out of the top 500 supercomputers, 86% are equipped with these shared-memory compute cores [2] (data for Nov 2008). Researchers have indicated that for the NAS, NAMD and HPL parallel benchmarks, on an average, about 50% of the total byte-volume of data transferred is through intranode communication [10], i.e., between cores on the same chip and also between cores on different chips but on the same node.

Extending the processor micro-architecture to provide support for message passing between cores is now a viable option. Shrinking process technologies have increased transistor densities per chip. Since individual cores are unlikely to increase in complexity due to growing concerns on power consumption, a part of the excess transistor budget can be allocated for auxiliary hardware structures to facilitate message passing. Since this messaging-controller would be designed for intra-CMP communication and it does not have to deal with communication with processes running on distant nodes, its complexity is limited by the number of processing cores on the chip.

A further motivation for the inclusion of hardware accelerators for MPI primitives is the relative inefficiency of existing software based techniques. We take a look at the drawbacks and overheads of the techniques described in related work in the following section.

4.1.1 Drawbacks of software based optimizations

Mutiple copies The optimizations described in Section 2 depend predominantly on a dual-copy mechanism for message transfer - a copy out of the private data structure of the sender into a buffer in the shared address space and a subsequent copy from the buffer into the receiver's private area. The kernel assisted copy method eliminates one of these copies. But the overheads of the system calls needed for updating the page tables of the OS (for mapping the sender/receiver private area to the OS address space) and also for the actual copy are prohibitive and make this method worthwhile only for small messages.

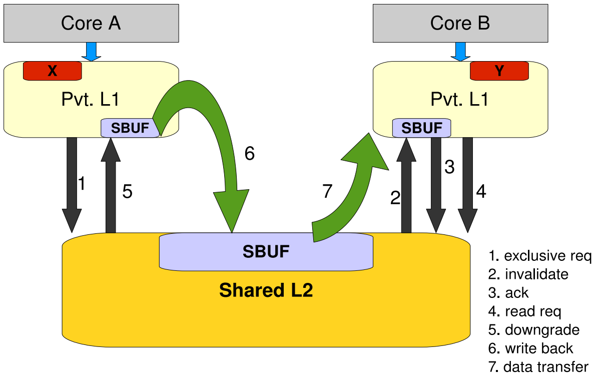

Figure 1. Cache Coherence Transactions

Cache Pollution The multi-copy mechanisms also have significant memory footprints owing to buffers needed for communication between each pair of probable sender and receiver. This in turn leads to cache pollution. While writing and reading from the shared buffer, it has to be brought into the L1 of the sender and receiver respectively. However there is no reuse of the data in the buffer and if the buffer is large enough, it can evict frequently used lines from the cache. This causes cache pollution that can adversely affect the computation phases of the application and consequently slow down the entire application.

Cache coherence The enhanced buffer management scheme and nemesis communication subsystem (described in Section 2) try to reduce the memory requirements and cache pollution issues by intelligent buffer organization and cache cognizant data arrangement. But a closer look at any scheme that involves a shared buffer reveals the unavoidable cache coherence traffic that it generates. Consider the scenario (shown in Figure 1) where core A does a send of the message stored in its private data space X to core B's private data space Y, using the shared buffer SBUF. Each core has a private L1 cache which is backed up by a shared L2. The sequence of read and write instructions issued by A and B to accomplish the transfer and the resulting cache coherence traffic can be summarised as follows :

- core A issues LD X

- core A issues ST SBUF

  core A requests for exclusive permissions on SBUF
  invalidation to core B for shared cache line in L1 containing SBUF
  acknowledgement of invalidation from core B

- core B issues LD SBUF

  core B requests for SBUF in shared state
  core A downgrades the status of the cached copy of SBUF to shared
  write-back of SBUF from core A's L1 to L2

- core B issues ST Y

So even a simple send-receive can lead to a large number of coherence messages - and this number would increase with message size and obviously with the number of transfers. If a message spans across several cache lines, and a single large shared buffer is used for the transfer as in the naive implementation, the coherence traffic will be the worst. However even in an implementation where the message is split into smaller segments, the problem is significant. Moreover, a large part of MPI traffic is comprised of collectives, i.e., broadcast like transfers. In such a case the number of send-receive operations scales with the number of cores and the coherence traffic is proportionately affected.

4.2 Proposed Hardware Messaging Controller

We advocate using a messaging controller per core that is responsible for controlling the message passing operations. Our intent is to reduce/eliminate the overheads associated with shared buffers and free up the main core from performing each individual load and store.
The objectives of the controller are to

- accelerate message transfer
- reduce redundant cache coherence traffic
- lead to better cache usage
- free the main core from executing each load and store (similar to the role of a DMA)

4.2.1 Functional overview of the messaging-controller

The basic mechanism of the transfer comprises of copying data from the sender's address space to the receiver's private space directly. To illustrate the desired functionality of this controller, we will look at the example in the previous sub-section where processor A sends the data from its private space X to core B's private data structure Y.

1. **On a send, X is brought into L2.** If the L1 cache has a write-back organization rather than a write-through one, the controller needs to do a look-up of core A's L1. If the copy of X in L1 is dirty, then a write-back of the data takes place into L2. If the copy is clean, then L2 has the up-to-date copy. On a L1 miss, the L2 is looked up and if necessary the data is brought into L2 from the lower levels of the memory hierarchy following regular caching rules. If the L1 follows a write-through scheme, then there is no need to do the L1 lookup.

2. **On a receive, Y is brought into L2.**. To prevent the core from reading a stale value of Y while the controller is carrying out the transfer, any cached copy of Y in B's L1 is invalidated. A miss in L2 lookup for Y would result in the block to be brought into L2 from the lower levels of the hierarchy.

3. **Copy from X to Y.** The controller issues a series of reads and writes to X and Y respectively in the L2 to accomplish the final transfer. This can proceed in parallel with any computation the cores are executing if it does not involve X or Y.

4. **Prefetch.** On completion of the copy, the controller prefetches parts of Y into core B's L1 as it is likely that core B would access the data in Y in the near future.

4.2.2 Asynchronous Communication

In the procedure outlined above, since there is no intermediate buffer, core A can modify the contents of X only after the transfer is complete. The mechanism is thus best suited for synchronous communication, i.e., when computation can advance only on successful completion of communication.

In any message passing system, there is also provision for asynchronous blocking and non-blocking communication [14]. Asynchronous mode allows an application to do a send-receive and proceed with computation (potentially modifying/using the private send-receive data structures) before the entire communication is completed. In the blocking case, the computation can proceed only when the process receives a confirmation that the data has been buffered by the system, while in the non-blocking case, theoretically the computation can proceed immediately after posting a send or receive, but the transfer is carried out at an undetermined time.

After posting an asynchronous send, the core A might choose to reuse the data area X. Meanwhile, the matching receive operation by core B might not have been issued. This necessitates buffering the data so as to avoid violating the semantics of the send operation.

Approach 1 We propose to do the buffering directly in L2, bypassing the L1s of either core. Thus after steps 1 and 2 in the transfer procedure, the data from X is copied into the shared buffer and the control is returned to the sender. This avoids L1 cache pollution and also eliminates the cache coherence traffic as before.

Approach 2 We can also stall the processor after the asynchronous send and allow forward progress only when a matching asynchronous receive is encountered. Though this does not affect the correctness, the coerced synchronous behaviour may be unacceptable in some cases - where communication-computation overlap is desired.

Out of the different send modes provided by the MPI standard [1], the two most widely used are MPI_Send and MPI_ISend - but they do not need to allow immediate reuse of the send buffer. While MPI_Send is free to be blocking/non-blocking or even wait for a matching receive before returning control, the MPI_ISend semantics are such that a reuse of the send buffer is allowed only after receiving confirmation through a successful MPI_Wait and MPI_Test call to ensure that the data has been copied out of the send buffer. This allows us to use our default bufferless transfer mechanism in both cases.

4.2.3 Collective communication

In MPI parlance, collectives refer to communication routines that do many-to-one and one-to-many communication. There are a number of such routines viz. MPI_Bcast, MPI_Gather, MPI_Allgather, MPI_Scatter and MPI_Alltoall, each of which is comprised of multiple point to point messages. In our scheme, each MPI collective would be handled at the granularity of the individual transfers that make up the call. For example, during a MPI scatter, the root process sends each segment of a message to one of the receivers. Thus the message passing hardware has to carry out the transfer operation with the root process as the sender and each other process as the receiver. For a pair of processes, the copy would take place in the highest level of the memory hierarchy that these share. During collective operations, reuse of the send buffer is generally allowed only after the message transfer is completed - hence the controller can operate in a bufferless mode.

Since a collective would entail copying out various fragments of the same sender data structure into various receiver data areas, it might be necessary to have a pipelined architecture for the message-passing hardware.

4.3 Proposed implementation

As mentioned before, we plan to introduce one message-passing controller per core. The controller is invoked by the MPI library during a send-receive operation through new instructions that augment the ISA. On being provided with the send-receive buffer's address, the controller does the necessary cache lookups and sets up the data areas in the shared cache level by issuing non-temporal store operations to avoid cache pollution. The MPI library is responsible for setting up the controllers with the addresses, and also co-ordinate the send and receive controllers during the copy operation - only one controller would carry out the actual copy.

As is evident from the above discussion, the implementation would require instrumenting the MPI libraries with the special instructions besides extending hardware support.

Note that the innovations in Sections 3 and 4 are orthogonal and could be employed independently. The second innovation explicitly forces the copy to happen in the L2 and prevents the L1 caches from getting polluted by the intermediate buffers. This is consistent with what the first innovation attempts to do as well. Hence, both innovations can be combined. By eliminating L1 caching (with the first innovation) for shared data structures such as the intermediate buffers within the MPI run-time, part of the goals of the second innovation are automatically achieved.

Though in our initial design, we advocate the use of a single controller per core, we intend to explore other design alternatives by varying the number of cores served by a single controller and also by having one controller for sends and another for receives. To evaluate our design, we plan to make use of the SIMICS system level simulator to model the hardware innovations. We plan to choose MPICH2 having the Nemesis communication subsystem for intra-CMP message transfers as our baseline MPI implementation because Nemesis is regarded as the lowest latency mechanism for inter-core message passing.

5. Conclusions

This paper puts forth a number of innovations that are targeted at optimizing the execution of MPI workloads on aggressive multi-core processors. These include a controller that is in charge of performing copies, the elimination of L1 caching for intermediate buffers, the elimination of cache coherence for the most part, and the L1 caching of read-only and private blocks. It is too early to draw conclusions regarding the improvements possible with the proposed innovations. The paper makes arguments for why we believe this design is reasonable. Our early estimates show that the most potentially disruptive change – the elimination of cache coherence – only has a moderately negative impact on the performance of certain classes of shared-memory workloads. We therefore believe that the proposed design has the ability to significantly reduce design complexity and the overheads of cache coherence, while significantly boosting the performance of inter-core messaging and yielding tolerable slowdowns for the few shared-memory codes that may execute on such a processor. Much future work remains to quantify the benefits of the proposed architecture.

References

[1] The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/.

[2] Top 500 Supercomputers. http://www.top500.org/stats/list/32/procgen.

[3] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *Proceedings of HPCA*, 2009.

[4] B. Falsafi (Panel Moderator). Chip Multiprocessors are Here, but Where are the Threads? Panel at ISCA 2005.

[5] B. Beckmann, M. Marty, and D. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of MICRO*, 2006.

[6] D. Buntinus, G. Mercier, and W. Gropp. Data Transfers between Processes in an SMP System: Performance Study and Application to MPI. In *Proceedings of the 2006 International Conference on Parallel Processing*, 2006.

[7] D. Buntinus, G. Mercier, and W. Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, 2006.

[8] D. Buntinus, G. Mercier, and W. Gropp. Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing*, 33(9):634–644, 2007.

[9] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[10] L. Chai, Q. Gao, and D. K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[11] L. Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006.

[12] L. Chai, P. Lai, H. Jin, and D. K. Panda. Designing an Efficient Kernel-Level and User-Level Hybrid Approach for MPI Intra-Node Communication on Multi-Core Systems. In *Proceedings of the 37th International Conference on Parallel Processing*, 2008.

[13] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of MICRO*, 2006.

[14] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[15] P. Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of HPCA-11 (Industrial Session)*, pages 258–262, 2005.

[16] H. Jin, S. Sur, L. Chai, and D. K. Panda. LiMIC: support for high-performance MPI intra-node communication on Linux cluster. In *Proceedings of ICPP-05*, 2005.

[17] H. Jin, S. Sur, L. Chai, and D. K. Panda. Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, 2007.

[18] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of ISCA-24*, pages 241–251, June 1997.

[19] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of ICS*, 2003.

[20] N. Madan, L. Zhao, N. Muralimanohar, A. Udipi, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell. Optimizing Communication and Capacity in a 3D Stacked Reconfigurable Cache Hierarchy. In *Proceedings of HPCA*, 2009.

[21] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[22] N. Muralimanohar and R. Balasubramonian. Interconnect Design Considerations for Large NUCA Caches. In *Proceedings of ISCA*, 2007.

[23] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of MICRO*, 2007.

[24] NVIDIA. NVIDIA Tesla S1070 1U Computing System. http://www.nvidia.com/object/product_tesla_s1070_us.html.

[25] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, Network-Based Processing*, 2009.

[26] Tilera. Tilera Tile64 Product Brief. http://www.tilera.com/pdf/ProductBrief_Tile64_Web_v3.pdf.

[27] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of ISSCC*, 2007.

[28] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of ISCA*, 2005.