MOPED: Orchestrating Interprocess Message Data on CMPs

Junli Gu[1,2] Steven S. Lumetta[2] Rakesh Kumar[2] Yihe Sun[1]

[1]Institute of Microelectronics
Tsinghua University
Beijing, China Illinois, USA
junligu@illinois.edu
sunyh@mail.tsinghua.edu.cn

[2]Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
lumetta@illinois.edu
rakeshk@illinois.edu

Abstract

Future CMPs will combine many simple cores with deep cache hierarchies. With more cores, cache resources per core are fewer, and must be shared carefully to avoid poor utilization due to conflicts and pollution. Explicit motion of data in these architectures, such as message passing, can provide hints about program behavior that can be used to hide latency and improve cache behavior. However, to make these models attractive, synchronization overhead and data copying must also be offloaded from the processors.

In this paper, we describe a Message Orchestration and Performance Enhancement Device (MOPED) that provides hardware mechanisms to support state-of-the-art message passing protocols such as MPI. MOPED extends the per-processor cache controllers and coherence protocol to support message synchronization and management in hardware, to transfer message data efficiently without intermediate buffer copies, and to place useful data in caches in a timely manner. MOPED thus allows full overlap between communication and computation on the cores.

We extended a 16-core full-system simulator based on Simics and FeS2. MOPED interacts with the directory controllers to orchestrate message data. We evaluated benefits to performance and coherence traffic by integrating MOPED into the MPICH runtime. Relative to unmodified MPI execution, MOPED reduces execution time of real applications (NAS Parallel Benchmarks) by 17-45% and of communication microbenchmarks (Intel's IMB) by 76-94%. Off-chip memory misses are reduced by 43-88% for applications and by 75-100% for microbenchmarks.

1. Introduction

The challenge of effectively programming chip multiprocessors (CMPs) has received significant attention from both academia and industry. Most of these chips leverage manufacturers' understanding of hardware-coherent symmetric multiprocessor (SMP) architectures to provide similar support. Most CMPs today have small private L1 or L2 caches but share a large last-level cache that is kept coherent with all L1 caches. As the number of cores on chip increases, memory resources per core decrease, making it critical to design and program future CMPs to share memory efficiently. For CMPs to continue to scale, applications must be able to take advantage of parallel execution, and data must be moved efficiently amongst the cores.

A recent study [3] reported that desktop software is not meeting this challenge: although CMPs have dominated the desktop market for several years, evidence suggests that even codes designed to be parallel rarely make use of more than a few cores simultaneously.

Heterogeneous cores offer one answer to this problem: sections of an existing code base can be rewritten for execution on a many-core accelerator such as a GPU, then wrapped with additional code to move data between the CPUs and the accelerator. Accelerator memory models are typically non-coherent, and the heterogeneity in hardware today implies a similar heterogeneity in memory models. In such an environment, partitioning code and managing data movement efficiently can be major challenges, and recent work has explored convergence and integration of the two memory models to simplify code migration [12] and pave the way for more parallel applications.

An alternative hybrid strategy is to retain homogeneous cores, but to allow the programmer to enhance performance through selective use of explicit message passing. Message passing has dominated high-performance applications for decades, in large part because it enables programmers to specify data motion explicitly. In a CMP, the chip can use this information to move data more efficiently than is possible when the information must be speculatively deduced from individual operations, as is often the case with shared memory codes. As illustrated by Figure 1, threads

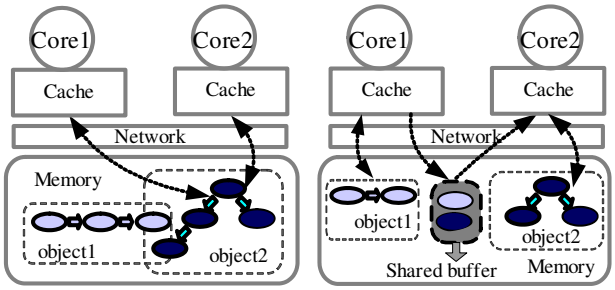

Figure 1. Illustration of shared memory (left) and message passing (right) models on a CMP.

using a shared memory programming share a single address space. Each thread can access the objects in the memory without notifying other threads, and coherence is supported by hardware. Communication occurs through synchronization primitives targeting a common object, such as a lock. Hardware often resorts to moving data from cache to cache on demand. In contrast, processes in message passing model typically use distinct address spaces. Communication then occurs by sending data via shared buffers in memory mapped into multiple address spaces. To leverage explicit communication, substantial previous work in the context of servers and high-performance computing has examined extension of cache coherence mechanisms to better support message passing, but few of the hardware mechanisms have been incorporated into today's CMPs.

In this paper, we explore this approach in the context of CMPs, developing a Message Orchestration and Performance Enhancement Device (MOPED) that offloads message synchronization and data copying from the cores and reduces message latency and coherence traffic through extensions to a directory-based MOESI protocol. MOPED provides support for general sender-receiver synchronization and data transfer within and across address spaces on a CMP. The MOPED design incorporates support for virtualization, allowing MOPEDs to be used simultaneously by many parallel applications. We evaluate MOPED through full-system simulation of a range of message passing benchmark codes on a 16-way CMP, explore the impact of several coherence extensions, and provide insight on the value and cost of the proposed coherence extensions.

The remainder of the paper is organized as follows. The next section reviews previous work on optimization of explicit communication. Section 3 illustrates sources of overhead in a widely-used message passing implementation for CMPs. Section 4 describes the basic MOPED design and implementation and estimates hardware complexity. Section 4.5 provides details of our coherence extensions. Section 5 describes our simulation infrastructure and the benchmarks we used to evaluate MOPED. Section 6 reports results and provides discussion, and Section 7 concludes.

2. Background and Related Work

Historically, message passing developed on distributed memory machines with separate operating system (OS) instances on each processor. Even when hardware coherence is available, models retain the use of private address spaces for each process both for portability reasons as well as to protect against asynchronous access to private data.

A large body of research work exists focusing on the problem of optimizing message passing and remote procedure calls on symmetric multiprocessors (SMPs) (e.g., [2]), and particularly on how to transfer data efficiently on hardware coherent shared memory. The primary strategies for achieving this aim are to reduce the number of copy operations, to transform processor overhead into latency by offloading work, and to hide latency through intelligent synchronization and/or adaptive cache policies.

Software-based message passing requires that processors perform all copy operations. Hardware assistance for moving message data was common in the era of massively parallel processors (MPPs) such as the Meiko CS-2, and is still used in networks based on Myrinet and Infiniband. One of the difficulties with offloading copy operations from processors to DMA engines is the need to support virtual memory (address translation). The microprocessor industry is only now standardizing support for generic devices. The potential for a processor-controlled translation unit for a network interface card (NIC) was explored in [18]. The processor handled page fault requests via interrupts and removed entries as necessary from the NIC memory as necessary. Some commercial systems, such as the IBM Cell processor, include DMA engines designed along similar lines for use with scratchpad memories [10]. Use of additional hardware thread contexts (as with Simultaneous Multithreaded, or SMT) avoids the translation issues but can interfere with the operation of the main thread of computation.

A comparison of mechanisms for shared memory messages appeared in [4]. As already mentioned, the most basic mechanism is a shared memory buffer, but even this method allows a range of queuing disciplines. Simplistic point-to-point queues are easy to implement but scale poorly. Support for one-sided communication (active messages) on such machines using lock-free, many-to-one queue algorithms was explored in [15]. The basic shared memory algorithms for MPI appeared around the same time [6], but the best current implementation uses lock-free algorithms such as MPICH's Nemesis device [4]. We use shared buffers as the baseline model when evaluating MOPED.

A second approach to message passing is to trap into the kernel, which can manipulate data in both address spaces and can thus copy the data directly. This approach can make use of ptrace or Unix sockets, but is most effectively implemented as a kernel module, which reduces the number of copies required but is typically too expensive for short mes-

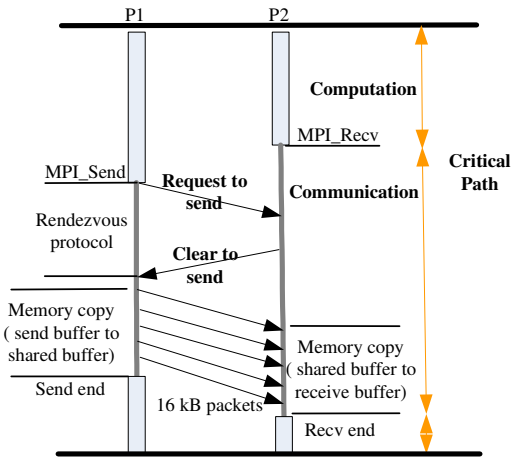

Figure 2. MPI rendezvous protocol.

sages. Page remapping is also theoretically possible, but message buffers are not generally page-aligned, and additional copies incur too much overhead.

Tailoring cache consistency protocols according to the type of data object being accessed was introduced by Munin [1], and numerous papers have explored application-specific protocols as well as ideas such as cache injection to reduce access latency for message passing. For example, one CMP design combines support for shared memory with support for streaming applications [14] uses a DMA engine to deliver data into a 24 kB scratchpad memory that in shared memory mode extends the L1 data cache.

Several recent papers have studied message passing in the context of CMPs. Some argue that CMPs should support only message passing [13], and Intel developed the Rock Creek architecture [7] to enable exploration of this idea. Although we evaluate MOPED's potential using MPI benchmarks, MOPED instead focuses on the potential for using explicit communication to enhance application performance without completely rewriting the code. A recent workshop paper [5] provided a high-level description of a hardware mechanism for optimizing message passing. We provide a more detailed discussion of practical implementation issues such as virtualization and hardware overhead as well as support for offloading message synchronization and address translation. Our MOPED reduces copy operations, reduces cache traffic, and frees processors from nearly all communication overhead. We also discuss mechanisms for optimizing control of moving data into and out of caches to benefit more general parallel programs. For comparison, the advanced DMA engine in the IBM Cell moves data between scratchpad memories near the cores. Rather than being managed dynamically and degrading gracefully with increased message length, as is MOPED, space must be reserved in these memories before message data are sent and remains reserved until the data are consumed.

Another study focused on accelerating data motion for staged execution models in the context of heterogeneous CMPs [17]. In a staged execution model, data flows from one process to the next in a pipelined manner. The paper describes mechanisms to identify these flows and extends the instruction set to allow each stage to specify the data to be moved. MOPED focuses on explicit communication rather than on specific producer-consumer models, which are more amenable to queuing.

3. Overhead analysis on CMPs

As mentioned earlier, implementations of the Message Passing Interface (MPI) standard typically leverage multiple address spaces both for portability and for inter-process protection of private data. In this section, we examine the path taken by a message passed from one address space to another in a hardware-coherent shared memory system and discuss sources of overhead incurred in this process.

Figure 2 illustrates the basic MPI protocol for sending a large message from one address space to another. Senders (P1) and receivers (P2) synchronize by exchanging control packets before copying message data via the shared memory region. This rendezvous avoids the deadlock scenario in which a large message fills the shared buffer and blocks transfer of a subsequent message that must be received before the first. The diagram illustrates the scenario in which the receiver arrives first. On entering MPI_Send, the sending process transmits a request-to-send control packet, and, in this case, receives an immediate reply. The sender then proceeds to copy the data from the send buffer into the shared memory region in 16 kB chunks. A blocking send returns only when all data has been copied into the shared memory region. A non-blocking send can compute during the time spent waiting for the initial rendezvous, but the processor itself must perform the memory copy. As illustrated by the figure, both synchronization and message data copying require CPU cycles, which increase the critical path of the program. As the number of cores increases, more communication between cores will be required.

As already mentioned, we chose the most widely-used message passing infrastructure, MPICH [11], as a baseline against which to compare MOPED. The MPICH shared buffer message passing mechanism (SHM) maps a shared-memory region into each MPI process' address space in order to transfer messages. Figure 3 shows a more detailed hardware view of message passing in our baseline system. Sending a message requires a dual-copy operation. The sender process copies data from the message's send buffer (Step 1) into the shared buffer (Step 2), bringing both buffers into the executing processor's caches. The receiver process then brings the shared buffer into its cache and copies the data from the shared buffer into the message's receive buffer. Step 3 of the figure shows the motion of cache lines for the shared buffer, and Step 4 shows write-

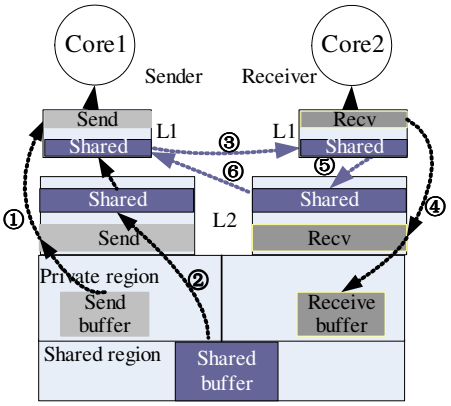

Figure 3. Dual copy message passing via shared memory buffers for original MPICH.

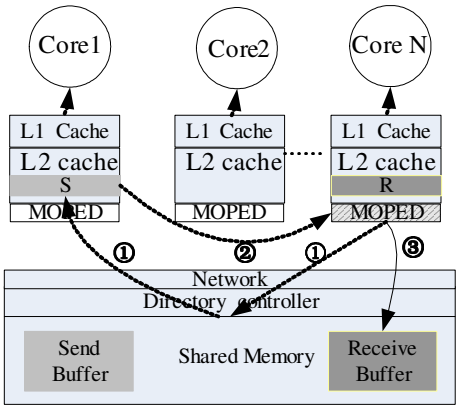

Figure 4. Zero-copy message transfer with MOPED.

back of the receive buffer. For a long message, the shared buffer lines may bounce several times between sender and receiver before the message transfer completes. The shared buffer is typically fairly large (256 kB) to avoid its becoming a bottleneck, thus it is usually evicted from the receiver's L1 (Step 5) before the sender reuses it (Step 6).

Several sources of overhead become clear from this overview. Software overheads arise from synchronization and from fragmenting the message into 16 kB packets. Messages exhibit a producer-consumer relationship, and most coherence protocols do not provide efficient mechanisms for managing them. All three buffers are brought into the L1 and L2 caches from memory, generating substantial cache traffic. Computation data are evicted from the caches due to cache pollution by these buffers. The shared buffers are significant, as they are large and allocated separately for each pair of communicating processes. Hardware coherent architectures provide coherence for all data, and message data thus suffers increased latency, cache traffic, and cache misses [4], which are obstacles to application performance.

4. MOPED Design

In this section, we discuss the design of a Message Orchestration and Performance Enhancement Device (MOPED). MOPED consists of simple extensions to each core's L2 cache controller, enabling message synchronization and transfer to proceed in parallel on all MOPEDs. With MOPED, message passing requires only a few loads and stores to deliver basic message information and to poll for completion. MOPED synchronizes message senders and receivers independently, then handles data transfer by interacting with the directory controllers and the local caches, leaving processors free to perform other work. MOPED moves data directly from the send buffer to the receive buffer, eliminating overheads associated with shared buffers. Use of MOPED requires changes to message passing library code, which we illustrate below. Application code that makes use of message passing libraries or runtimes need not be changed to use MOPED.

Message data transfer using MOPED with MPICH appears in Figure 4. The MOPED associated with the receiver process begins (Step 1) by requesting a copy of send buffer cache lines from the directory controller. These lines are usually in the cache associated with the sender process, but may also be in memory. In Step 2, a copy is delivered to the receiving MOPED, but the lines are not placed in the associated cache. Instead, MOPED writes the data directly into the receive buffer (Step 3), which is brought into the local cache since the receiver process is likely to access the receive buffer after the message has arrived.

4.1. MOPED software and virtualization

MOPED is designed to interact directly with user-level applications and must thus support virtualization of resources across multiple applications and processes. A parallel program first obtains MOPED resources from the operating system (OS) in the form of one or more pages that map to physical addresses assigned to MOPED. The OS maps the page or pages for any given program into all address spaces associated with that program (but not into any other program's address spaces). The program is then responsible for assigning each communicating agent, such as an MPI process, a unique 8 B region to be used as a data port and a poll port to MOPED. When a process loads or stores to these ports, the address used conveys both a program ID and a unique process rank to MOPED.

MOPED stores information about each message sent or received in a message descriptor (details in Sec. 4.2). From a process' point of view, a descriptor is represented by a small integer, a descriptor ID. A send or receive operation begins by requesting a message descriptor: a load from its MOPED data port returns a new descriptor ID. The process then provides information about the message by storing values concatenated with the descriptor ID to its MOPED data port. Once all necessary information has been stored, the process can proceed with other work. The process can

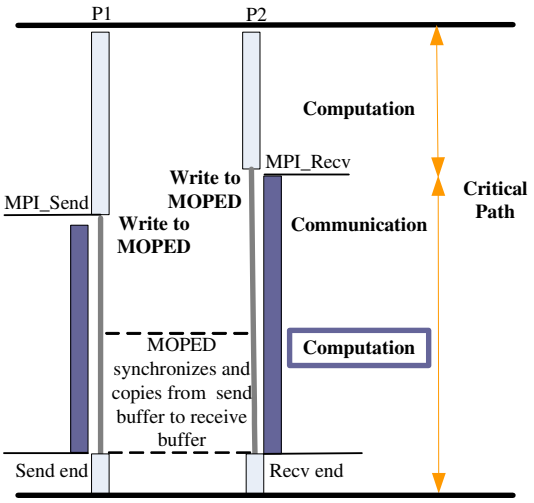

Figure 5. MPI protocols working with MOPED. Non-blocking send/receive allows computation during message transmission.

check for completion of the message at any time by storing the message's descriptor ID to the process' MOPED poll port and then loading from the same port.

Message passing applications often prevent process migration, but MOPED guarantees correct execution despite interruptions by the OS scheduler and regardless of migration. Towards this end, descriptor IDs encode both the local MOPED's ID as well as an index specific to that MOPED. Even if a process migrates immediately after obtaining a descriptor ID, the data from stores needed to complete the message descriptor as well as loads for polling purposes can be forwarded to the original MOPED. An additional mechanism is necessary to protect against context switches during the poll store-load sequence. Such events are rare, so each MOPED caches only a single mapping from a program ID/process rank to a descriptor ID. If a process polls using a port that does not match the cached value, MOPED returns a specific value indicating that the poll must be retried. In response, the software re-executes the poll port store to refresh the cache, then re-executes the load.

For evaluation purposes, we integrated MOPED into the MPICH runtime, capturing and emulating the port operations within FeS2. No changes were necessary to MPI application code. We modified blocking and non-blocking sends and receives, message completion (MPI_Wait), and all collectives. The result is illustrated in Figure 5. Processors avoid the need for software rendezvous by writing a message descriptor to MOPED. Blocking sends and receives poll immediately for completion, whereas non-blocking calls return and use MPI_Wait to poll at some later time. Freeing the CPU from communication enables better overlap between computation and communication.

Outstanding descriptor

Status RVNMAC	Prg ID procRnk	Desc ID	Page table ptr	Send /recv	Buf info	Con -text	rank	tag

RVNMAC: Reserved, Valid, Needs match, Matched, Active, Completed

Matched descriptor

Sender info			**Receiver info**		
Descriptor ID		Send_buf vir_Addr	Descriptor ID		Recv_buf vir_Addr, size
2X phys page addrs		Page table ptr	2X phys page addrs		Page table ptr
page1	page2	CR3	page1	page2	CR3

Figure 7. MOPED message descriptors.

4.2. MOPED hardware

The MOPED hardware (Figure 6) consists of three main components: a table of outstanding message descriptors, a table of active matched messages, and a copy control unit that translates addresses, interfaces with the L2 cache and directory controllers, and handles cache line re-alignment.

When a process requests a descriptor, the MOPED associated with the core executing the process (the "local" MOPED) finds a free entry in its outstanding descriptor table, marks that entry as reserved, and returns an ID for it. The process then writes the descriptor content into the local MOPED's table. As shown in Figure 7, each descriptor contains information about the message buffer (virtual address and length) and information for message matching (communication context, message tag, and process rank within the communication context [11]). MOPED also records whether the descriptor corresponds to a send or a receive, the process' page table pointer (in x86, the page directory base register, or cr3) as well as the program ID and process rank that made the request, which ensures that other process' descriptors cannot be accessed accidentally (or maliciously). The descriptor ID field serves for forwarding, linking matched descriptors, and other linking purposes.

When a process completes a descriptor, MOPED must try to find the matching descriptor. However, the two descriptors must first be within a single MOPED. Each descriptor contains the program ID and receive process rank for the message. These two are hashed to select a MOPED at which the descriptors can be matched, and the local MOPEDs forward the descriptors through the on-chip network to the "matching" MOPED identified by the hash.

The matching MOPED marks each descriptor received in this fashion as valid and as needing a match check. MOPED's message matching adopts MPI's generic matching scheme, which was developed to generalize a wide range of preceding message passing systems. Two descriptors must come from the same program to match, and must have opposite types: one send and one receive. They must also match in context, tag, and sending process rank, but both tag and sender rank can be matched using wildcards in the receive descriptor. Implementing matching is straightforward: a simple state machine starts by finding the first descriptor that needs a match search, then walks through

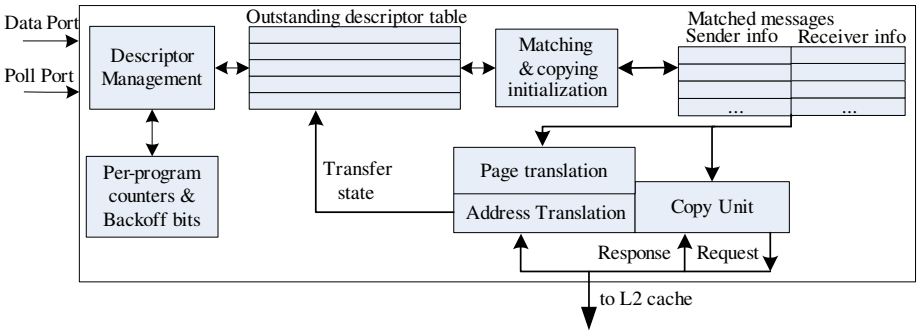

Figure 6. MOPED block diagram.

the outstanding descriptor table searching for a matching descriptor. If no match is found, the descriptor's status bit requesting a match check is cleared—the matching descriptor will appear later. Once a match has been found, the pair of descriptors is forwarded to the MOPED local to the receiver process and marked as matched.

Matched descriptors are moved into the active match table in a manner similar to that used for matching. When a table entry is free, the first descriptor with a match bit set and its matching descriptor are brought into the table and marked as active. The table entry includes information from both matched descriptors (see Figure 7). As described below, MOPED copies data for active matches. When a data transfer completes, the descriptors on both local MOPEDs are marked as complete and left until the processes poll them. When a process polls a completed descriptor, the descriptor is freed for reuse.

The copy unit performs data transfer for each matched message. Each active match table entry contains information for address translation and copy management. We use only a single copy unit, but distinct units could enable more flexible copy policy. Figure 8 shows a copy unit, including two address generators, a merge unit, cache request management, and write back logic.

During copying, an address generator for each buffer generates individual cache line addresses. Virtual to physical address translation is handled in a pipelined fashion in parallel with copying. In practice, double-buffering suffices, since a new translation is only necessary for every page of copying. Translation of the first two pages in the buffer could start as soon as a descriptor is valid. MOPED walks the page tables to obtain physical page addresses. A staging buffer of a single cache line suffices for walking page tables, but a slightly larger set can leverage spatial locality in page table data, since buffers are virtually contiguous. After a page has been copied, its physical address is discarded, and a new translation obtained for the buffer page after the page currently being copied. If a page translation is found to be missing, MOPED interrupts the associated processor and asks the OS to map the page into memory.

Send and receive buffers can have arbitrary alignment,

	Addr	State	First	Last	mark	leftCopy	rightCopy	Data
bits	64	2	1	1	1	1	1	Cache_Line_size

Figure 9. Send/receive line structure in copy unit.

thus MOPED must re-align data when copying. As seen in Figure 9, send and receive lines in the copy unit are cache lines extended with a physical address, a state, and copy control bits. Lines resident in MOPED are logically part of the L2, and tags must be checked on L1 misses. States include Free, Reserved, Requested, and Present. Reservations are made in Free lines by assigning a physical address in a cyclic manner. If a coherence operation is issued to fill the line (send buffer) or to gain exclusive access rights (receive buffer), the line becomes Requested. Once the coherence operation completes, or if the line is already present in the local cache, the state becomes Present. Data are copied from Present send buffer lines to Present receive buffer lines. A receive line may move from Present back to Reserved if the cache controller needs to revoke exclusive access rights to the line. The merge logic copies the data from send lines, then packs and aligns them to the receive lines. Each send line shifts its data to receive lines using a fixed left and right shift number (not shown, but stored with the merge logic). The send/receive lines work as a cyclic queue. The write back logic then writes full receive lines into the memory. Once data have been copied from a send line, its state returns to Free. An analogous transition occurs to receive lines once their data has been written back into the cache.

To control the copy process, flag bits (First and Last in Figure 9) are used to record whether a line is the first or last line in a buffer and whether the left or right halves of copying are finished. A cycle mark is used to ensure that copying from send to receive lines only occurs when the lines are in the same cycle of use.

4.3. MOPED coherence protocols

We extended the directory-based MOESI protocol in Ruby [16] to support requests from MOPED. Since MOPED is co-located with an L2 cache, We move any cache lines from the L1 into the L2 before operating on

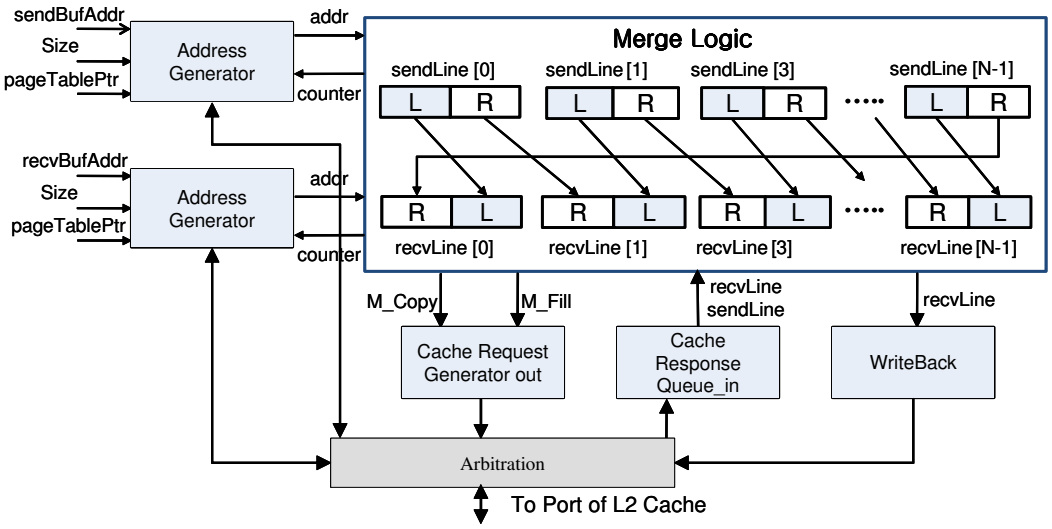

Figure 8. MOPED copy unit. Send and receive lines are logically part of the L2 cache. N=4 in our evaluation.

them with MOPED. The directory controllers do not distinguish between MOPED and the L2, and the basic MOPED approach requires no changes to the directory controller. However, simple extensions to the directory controllers can further reduce overhead and pollution (see Section 4.5).

For cache lines in a send buffer, MOPED requires read access (GETS), but retaining a copy in the receiver's cache merely pollutes it. To differentiate requests from MOPED from processor requests, we added an event, M_GETS, to request read access from the L2 cache controller, and an extra transition state, ID, that indicates that the cache controller is waiting for the return of data from a GETS generated by MOPED. The directory controller sees no difference between read requests (GETS messages) generated by the processor and those generated by MOPED. When data for an M_GETS arrives, it is stored in the copy unit. If exclusive access has been granted by the directory controller, the line is immediately written back to memory (as if evicted from the L2). Otherwise, no further action is necessary.

For cache lines in a receive buffer, MOPED requires exclusive access (GETX) in order to fill them with data. In this case, maintaining a copy in the receiver's L2 is likely to be beneficial. The baseline design thus needs only a new event, M_GETX, to ensure that MOPED is notified when exclusive access has been granted.

4.4. Deadlock avoidance

The MPI runtime can deadlock if the finite resources available for matching messages are filled with messages that do not match. This property is theoretically unavoidable with finite resources and arbitrary code, but MOPED may exacerbate the problem by sharing resources among many message passing programs.

To avoid such problems, MOPED can force the runtime to back off into using the original software mechanism for message passing. The easiest way to accomplish this goal is to track the number of outstanding descriptors for each program. When MOPED must refuse a descriptor reservation, it sets a backoff bit for that program on all MOPEDs. Reservation requests check this backoff bit for the corresponding program and, if it is set, fail regardless of whether or not space is actually available in the table. Descriptor poll operations also check the backoff bit. When it is set, and the polled descriptor has not already been matched, the descriptor is immediately removed from the table. The runtime must then revert to the original software scheme. Revocation is necessary to avoid having matching descriptors split between MOPED and the software mechanism, in which case they can never be matched. Once all descriptors for the program have been removed from the table, either by revocation or completion of the message transmission, the count of outstanding descriptors for the program reaches zero and the backoff bit for the program is cleared. The backoff bit is visible to the program, and a software barrier is needed to synchronize process ranks before again using MOPED.

4.5. Coherence Protocol Optimizations

As we see in Section 6, the baseline MOPED design produces fairly good improvements in both performance and coherence traffic, but for maximal benefit, we must also extend the directory controller design.

We apply two optimizations towards this end: copy optimization for send buffer lines, and fill optimization for receive buffer lines. A send buffer line must merely be copied at some instant in time, thus making modifications to the current cache state is unnecessary, and is in fact often detrimental to performance. In particular, send buffers are often reused, and removing them from the sender's cache forces the cache to re-load them later before refilling them with data. The copy optimization thus enables MOPED to re-

quest the data for a cache line without obtaining any access rights to the line. If the line is held exclusively in a remote cache, it remains there in a modifiable (or modified) state. If the line is off the chip, it is brought in but not retained other than in the MOPED copy unit.

The second optimization applies to receive buffer lines. As MOPED fills such lines completely with new bits, there is no need to fetch the old data. Old copies of a line must still be invalidated, but none of the line's data need be forwarded to MOPED. Instead, a line filled with 0 bits is placed in the cache associated with the receiver's MOPED in a locally modified (dirty) state.

We now introduce the three configurations of MOPED that we use in the evaluations in the next section. The baseline design, BASEMOPED, extends the original MOESI cache controller implementation with support for MOPED requests, but does not change any of the original transitions. As a result, send buffer data is flushed from the caches when the receiving MOPED reads it. The two other designs both fix this problem by changing a single cache controller transition in order to retain a copy of the send buffer data in the sender's cache. Each of the remaining designs adds a new coherence message type and a few transition states to the directory controller to support MOPED more efficiently. The OPTCOPY design enables MOPED to obtain a copy of a send buffer line without changing the current state of the line, and the OPTCACHE design extends OPTCOPY to grant exclusive rights to receive buffer lines without first obtaining a copy of the current data (invalidations are still performed). Serialization of both new operations occurs at the owner or the directory.

4.6. Hardware complexity

The MOPED design leverages existing hardware to limit added complexity. For example, copy operations execute using an extended coherence protocol and relying on the L2 cache controller to handle the bulk of the work. Address translation logic to walk page tables in memory can be extracted from the core design, with page faults handled by a core (as in the IBM cell [10]). Finding and executing matches uses find-first-bit and linear table walks. Merge logic requires only shifts.

Unmatched descriptors require roughly 24 B of storage, and a table of 64 suffices. The active match table need only be big enough to hold the number of matches that can execute in parallel. In our simulations, we allowed only a single match. The entry itself occupies about 70 B, and the associated copy unit occupies 71 B for each send/receive line (we used four of each). The total storage for these elements is just over 2 kB per MOPED, which is tiny compared to the L2 cache with which it is associated (or, for that matter, the L1 or branch predictor state in a high-end processor).

The overhead incurred for coherence state by MOPED is essentially negligible. The basic MOPED design adds two new events and a single transition state to the L2 cache controller. The new transition state handles requests for send buffer lines, which are not placed in the cache on arrival. Only a small number of cache lines can be in transition at any time; these are stored in miss status handling registers (MSHRs, or TBEs in the Ruby code). MOPED does not add new (non-transition) states for lines in the cache, and thus no additional storage is needed for the caches.

Neither of the more advanced MOPED designs adds more states to the cache controller. The OPTCOPY design adds four new transition states to the directory controller, and the OPTCACHE design adds a fifth. Given the number of transition states in GEMS' original directory-based MOESI protocol (4 bits for 12 states), the only design that requires additional state bits is OPTCACHE, which adds one bit per line that can be in transition per directory controller. No non-transition states are added to the directory controller by any version of MOPED, thus no extra state bits are needed for directory entries.

Table 1. Simulated Infrastructure

Infrastructure	Description
System	Cache-coherent CMP
OS	Fedora5 (linux2.6.15)
S/W	MPICH-2-1.2
Cores	16× Pentium 4, 1GHz
L1 I/D Cache (Private)	32 kB, 4-way, 64-byte line size 1-processor-cycle latency
L2 Cache (Private, exclusive of L1s)	16×512 kB, 8-way, 64-byte line size, 10-processor-cycle latency
Coherence Protocol	Directory based MOESI
Main Memory	512 MB, 35-cycle latency
Network Topology	Hierarchical Switch, 500 MHz
MOPED Parameter	**Description**
Coherence Messages	2 ops/cycle
Copy Unit	4 send + 4 receive lines

5. Infrastructure and Benchmarks

We have built a cycle-accurate full-system simulator based on Simics and FeS2. Simics is a functional simulator capable of simulating an OS as well as all user applications executing on it [9]. FeS2 [8] is an accurate execution-driven timing-model that includes two modules: the cache hierarchy Ruby from GEMS [16], and Pyrite [19] for branch predictors and an out-of-order x86 Pentium4 core. Details of our infrastructure appear in Table 1. The simulator can be driven by scripts to switch between three modes: functional, warm up, and timing. Measurements are made in timing mode. Warm up mode performs detailed simulation, but only to warm up micro architectural structures such as caches and branch predictors. Functional mode allows fast execution of initialization code, including the OS boot sequence. Timing mode reports detailed statistics on proces-

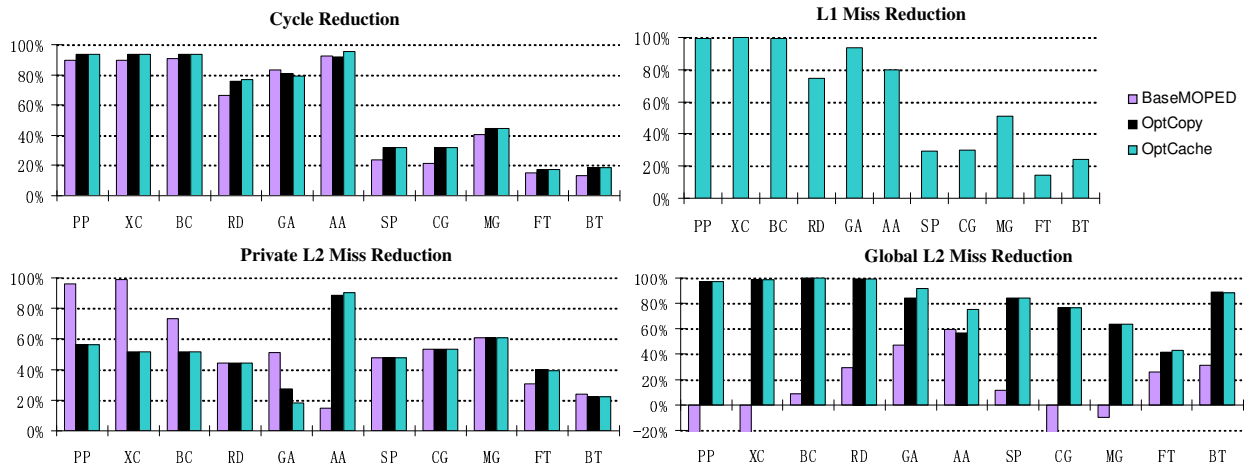

Figure 10. Reduction metrics with the BASEMOPED , OPTCOPY and OPTCACHE (relative to unmodified MPI).

sor cycles, cache behavior, and coherence traffic. We chose a private L2 architecture because of simulator limitations. However, the analysis and design in this paper apply to processors with shared memory, including cores that have shared L2 caches.

We make use of two benchmark suites to evaluate hardware potential as well as our MOPED design. Five of the size S NAS Parallel Benchmarks (NPB 2.4) serve as examples of realistic MPI applications. These include SP (scalar pentadiagonal), CG (conjugate gradient), MG (multigrid), FT (fast fourier transform), and BT (block tridiagonal). The second group of benchmarks, from the Intel MPI Benchmarks (IMB 3.2), represents basic communication patterns including Pingping (XC, an exchange), Pingpong (PP), Bcast (BC), Reduce (RD), Gather (GA), and All2All (AA). These primitives are common in message passing codes. For IMB, we measure 10 iterations with 16 kB messages.

6. Results and Measurements

In this section, we report our measurements and results. The baseline design in our comparisons executes unmodified MPI applications without MOPED. We compare the baseline against the three MOPED designs described in Section 4.5. As mentioned earlier, MOPED achieves optimal performance when each process in a program occupies a single processor and never migrates. We pin processes to processors for this purpose in our experiments, but the relationship between process and processor is recorded by MOPED whenever a process writes a descriptor.

The results include reductions in execution cycles and cache traffic including L1 cache misses, private L2 cache misses (these hit in another L2), and global L2 misses (serviced by memory). Figure 10 reports all results in four graphs. Each graph shows one type of reduction for all three MOPED designs normalized to the baseline. L1 cache miss reductions (upper right) were nearly identical for all designs, thus the graph includes only the OPTCACHE results.

For the IMB benchmarks, BASEMOPED achieves a 66-93% reduction in execution time and removes most L1 misses as well as most private L2 misses except for AA and RD. For the NPB, BASEMOPED achieves a 13-40% reduction in execution time while removing 13-48% of L1 misses and 24-61% of private L2 misses. In contrast, reductions in global L2 misses are not large, and PP, XC, CG, and MG see an increase in global L2 misses. The cause of this problem is an optimization performed by the original protocol: a read request to a locally modified line by another cache controller results in forwarding that line exclusively to the requesting cache. With MOPED, send buffer lines are copied and then written back to memory by the receiver's MOPED, implying that they must be pulled back in from memory in order to reuse the send buffer. As both microbenchmarks and applications routinely reuse send buffers, these programs show an increase in global L2 misses. A single change to the cache controller design suffices to eliminate these extra misses; this change is included in both of the more sophisticated designs, OPTCOPY and OPTCACHE.

The OPTCOPY design extends the directory controller to provide copies of send buffer lines without modifying other cache state. Relative to the BASEMOPED design, a significant number of global L2 misses become private L2 misses; in other words, data are retained in other caches on the chip, thereby reducing memory bandwidth requirements. In some cases, private L2 misses are still reduced overall compared with BASEMOPED. Execution time relative to the baseline is reduced by 76-94% for IMB and 17-45% for NPB.

The OPTCACHE design shows little improvement over the OPTCOPY design for either set of benchmarks. With relatively small messages and data sets, message data remain in the cache, and MOPED's copy optimization is enough to provide optimal behavior. Send buffer data re-

main dirty in the sender's cache, and receive buffer data remain dirty in the receiver's cache. MOPED copies one buffer to another by simply moving the data line by line from one cache to the other.

The benefits of MOPED's receive line fill optimization become clearer with larger messages. We ran the IMB benchmarks PP, XC, and BC with 1MB messages to evaluate these benefits. For PP and XC, the OPTCACHE design transforms roughly 50% of global L2 misses into local L2 misses corresponding to line fill operations. For BC, global L2 misses are reduced by 93%. The send and receive buffers are interlocked in terms of performance, thus neither PP nor XC sees a significant performance improvement: the send buffer is too large to fit in the sender's cache and must be retrieved from memory. BC, however, achieves a 5% reduction in execution time for OPTCOPY and a 22% reduction for OPTCACHE, both relative to BASEMOPED.

7. Conclusion

In this paper, we have investigated the potential benefit of providing hardware support for accelerating data transfer in the context of a chip multiprocessor. We described the design of a Message Orchestration and Performance Enhancement Device (MOPED) that operates through the on-chip coherence protocol to offload synchronization and data transfer overheads from processors, thereby freeing them to perform more useful work. MOPED manages sender-receiver synchronization and integrates with cache and directory controllers to enhance performance and to reduce both coherence traffic and pollution associated with message passing. Applications that use libraries or runtimes to support communication can leverage MOPED without change. As we showed with MPICH, only the runtime needs to be changed to use MOPED. Although we do not explore the possibilities in this paper, MOPED's integration with the coherence hardware also makes it possible to explore prioritization policies for message transfers, message data injection into upper-level caches. By overlapping communication with computation and offloading overhead from the processors, MOPED enables substantial performance gains. We believe that MOPED can make code migration easier by improving the programmer's return on investment for parallelization and by enhancing the gains possible with explicit message passing.

Acknowledgements

This work was made possible with the support of the NSF/IBM Blue Waters Project, NSF CCF, NSF CNS, NSFC, Intel, GSRC, Advanced Micro Devices, an Arnold O. Beckman Research Award, the Information Trust Institute of the University of Illinois at Urbana-Champaign, and the Hewlett-Packard Company through its Adaptive Enterprise Grid Program.

References

[1] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *Proceedings of PPoPP'90*, pages 168–176, 1990.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[3] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of ISCA*, pages 302–313, 2010.

[4] D. Buntinas, G. Mercier, and W. Gropp. Data Transfers between Processes in an SMP System: Performance Study and Application to MPI. In *Proceedings of ICPP*, 2006.

[5] N. Chatterjee, S. H. Pugsley, J. Spjut, and R. Balasubramonian. Optimizing a Multi-Core Processor for Message-Passing Workloads. In *Proceedings of the Workshop on Unique Chips and Systems (UCAS-5)*, 2009.

[6] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. In *Journal of Parallel and Distributed Computing*, volume 40, pages 35–48, January 1997.

[7] H. Howard and S. Dighe. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proceedings of ISSCC*, pages 108–109, 2010.

[8] http://fes2.cs.uiuc.edu/index.html.

[9] https://www.simics.net/.

[10] http://www.ibm.com/developerworks/power/library/pa cell.

[11] http://www.mcs.anl.gov/research/projects/mpich2/.

[12] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: a hybrid memory model for accelerators. In *Proceedings of ISCA*, pages 429–440, 2010.

[13] R. Kumar, T. Mattson, G. Pokam, and R. V. D. Wijngaart. *Multiprocesor System-on-Chip: Hardware Design and Tool Integration*, pages 115–123. Springer-Verlag, 2010.

[14] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *Proceedings of ISCA*, pages 358–368, 2007.

[15] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, California, November 1997.

[16] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005. *Proceedings of dasCMP*.

[17] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *Proceedings of ISCA*, pages 441–450, 2010.

[18] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-level Network Interfaces. In *Proceedings of HOT Chips*, 1997.

[19] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of ISPASS2007*, pages 23–34.