

plementation issues on PIMs. Initial analysis of this prototype has yielded some interesting results.

The remainder of this paper is organized as follows. The PIM computational model used for this work is presented in Section 2. This is followed by details of the MPI implementation for this model in Section 3. The evaluation methods used are discussed in Section 4, and some preliminary results are shown in Section 5. Section 6 presents a comparison to related work. Finally, conclusions and future work are presented in Sections 7 and 8.

2. Processing-In-Memory

Processing-In-Memory (PIM) [19, 20, 7, 30, 29, 28, 37, 18, 36, 8, 11, 10], also known as Intelligent RAM[34], embedded RAM, or merged logic and memory, combines both high speed logic and dense DRAM on the same die, thus exploiting the tremendous amounts of available on-chip bandwidth, and providing low latency access to a relatively large amount of local state. This serves to circumvent the von Neumann bottleneck, locally, and provides a unique challenge to the designer of large scale PIM systems. Specifically, there are a large number of processing resources available with high bandwidth, low latency access to their local memory. However, off-chip accesses suffer from the same high-latency, low-bandwidth problems that all modern parallel machines suffer, except that the pins that were previously wasted on caches and memory interfaces to dumb commodity DRAM can now be used for direct PIM to PIM communication, and can be designed to run at higher signaling rates as a result.. In the case of PIM, the disparity between these two types of memory access (local and remote) is significantly greater than other systems. Furthermore, because the systems under consideration consist of hundreds of thousands (to millions) of nodes, providing a petabyte or more of physical memory, the need for a simple, low overhead communication mechanism is paramount.

2.1. The Parcel Interface

The nodes communicate via the *parcel* (*PAR*allel *Com*munication *E*lement) interface [20, 36]. Parcels are messages possessing intrinsic meaning which are directed at named objects, somewhat similar to active messages [39]. Rather than merely serving as a repository for data, they carry distinct high-level commands and some of the arguments necessary to fulfill those commands. Low level parcels (which may be entirely handled by hardware) may contain simple memory requests such as: “access the value X and return it to node K ”. Higher level parcels are much more complicated. An example might be “begin execution of procedure Y with the following arguments and return the result to node L ”. A parcel is capable of performing

both communication and computation, and it may be generated by the user, run-time system, or hardware for whatever mechanism may be appropriate. Fundamentally, traveling thread parcels may also represent a thread continuation[17], and the memory system is capable of quickly relocating threads (via the parcel interface) implicitly, based on the memory addresses that a thread accesses, or explicitly, via a migrate request.

2.2. Traveling Threads

The *Traveling Thread* model of computation [29, 30, 28, 21] directly addresses the requirement for low-overhead support to co-locate computation and its required data by allowing extremely light weight threads to move between processing resources and consume the available local data without explicit programmer intervention. The parcel-based model of computation introduced the ability to remotely initiate threads at a memory node containing some data relevant to the computation being performed[36, 20, 7], and provides support for very fine grain determination of data locality, but required the programmer (or compiler) to control thread movement via remote method invocations (RMIs). The Microserver model minimizes the total amount of state transferred between nodes, but generally, the programmer (or compiler) explicitly identifies and transfers the intermediate state of the computation. (It should be noted that if the code is written in an object oriented fashion, automatic identification of the intermediate state is significantly easier.) In effect, computation occurs as a series of remote method invocations, each of which performs computation local to the PIM upon which it is running. Different degrees of thread state may be implemented on top of this model, from arguments to a procedure call to the automatic caching of the associated thread state[29]. Furthermore, different types of threads may be remotely invoked, ranging from an SPMD program, through the instantiation of a method, to position-aware traveling threads that explicitly move from PIM-to-PIM as its data needs change. For an MPI library writer, the encapsulation of state associated with traveling threads helps to reduce the complexity of state-full checks inside the library.

Traveling threads eliminate the logical overhead associated with maintaining coherency information among shared pieces of data, while simultaneously potentially reducing the physical (latency) overhead by converting two-way (remote data request) transactions into one-way (thread migration) transactions. Unlike traditional thread implementations, the spectrum of threads available on this class of PIM systems is extraordinarily light weight. A lightweight thread example could be the C statement, $A[i]++$ which could be a thread that moves to memory location $\&A[i]$ and increments the data there. Were that location a remote PIM,

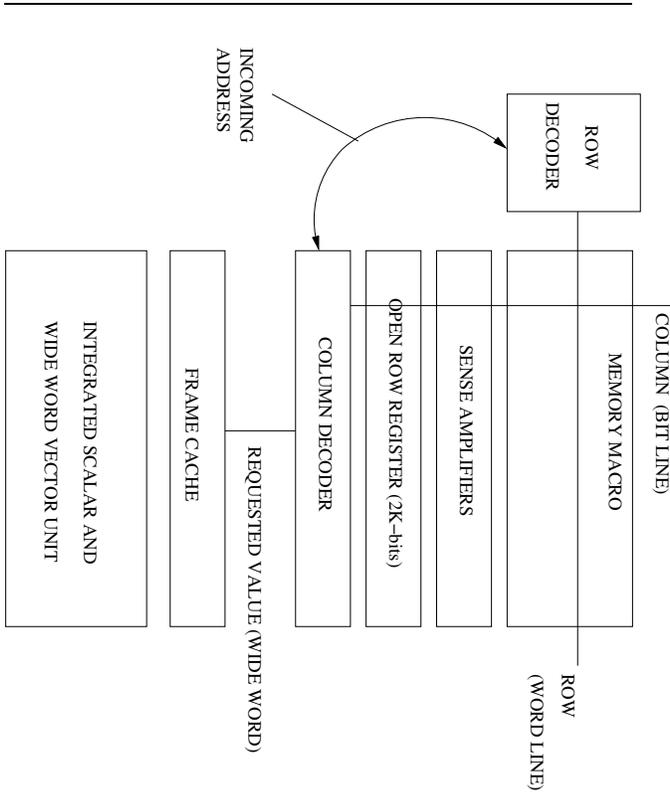


Figure 1: Typical PIM Node Architecture

and the contents of $A[i]$ unneeded in the next step of computation, a single, one-way traveling thread could be dispatched to perform the increment. These threads are particularly useful for scientific and data intensive codes which stream through memory quickly and show little temporal reuse. In the case of the simple example given above, A represents a large, shared global data structure, and i a piece of local state generated in the loop that’s required to carry out the trivial thread increment operation.

2.3. PIM Lite: A Typical PIM Node and Memory Layout

Figure 1 shows the typical layout of a PIM typified by the PIM Lite[6] or MIND[37] architectures. The processing resource are pitch-matched to the memory, providing scalar or short vector processing for a *wide word* (typically 256-bits) of data read directly from the open row register of one or more memory macros. Each PIM node provides processing from its local memory in this fashion. The simulated PIM nodes in this work reflect the PIM Lite architecture. (It should be noted that multiple PIM nodes may appear on a given PIM chip).

PIM Lite and the PIM execution model used in this

work support a multithreaded execution model that is heavily influenced by the early work in hybrid dataflow architectures including P-RISC [31] and Monsoon [33], as well as the Threaded Abstract Machine (TAM) [14, 13]. The lightweight messaging was also inspired by protocols developed for split-phase memory access in the dataflow work, as well as active messages [40], the MDP, [15] and J-Machine [32]. This prior work has been extended by adapting lightweight multithreading and communication for use with wide-words from on-chip memory and also integrates short SIMD operations into the architecture.

A PIM *node* consists of a block of physical memory with an associated processor. A collection of nodes interconnected on a network (independent of chip boundaries) is a *fabric*. Externally, the fabric appears as a single, physically-addressable memory system. Internally it operates as a distributed shared-memory multiprocessor, where each node can host multiple threads of execution from a single process, and each thread sees the same global shared address space.

In PIM Lite, nearly all the state information of a thread is taken out of the CPU and kept in memory at all times. In place of named registers in the CPU, thread state is packaged in *data frames* of memory. Logically, a frame is simply a region of contiguous memory locations within a single node. Physically a frame consists of one or more rows in a memory block. In PIM Lite, frames have a fixed size of 4 wide-words or 32 16-bit words, comparable to the size of a typical RISC register file. With the use of frames, virtually all instructions become operations on “memory” locations, expressed as operations on values within a single frame, between global memory and a frame, or between two frames. The frame cache allows fast access to this information, similar to a register file in a modern microprocessor. Unlike PIM Lite, the simulated PIM architecture provides a traditional RISC register file for each thread.

The execution state of a thread in the PIM Lite processor is completely described by two pointer values: a *frame pointer*, FP , which points to the starting location of the data frame, and an *instruction pointer* IP that points to the current instruction. Bundled together, the pair of values $\langle FP, IP \rangle$ is called a *continuation*, in the same sense as [33], and forms the basic unit of execution in PIM Lite. In the PIM Lite architecture, a thread pool stores continuations that are ready for execution. During an instruction cycle, a single continuation is removed from the pool and processed, and 0, 1, or 2 continuations are written back to the pool. The pool may contain multiple continuations with the same frame pointer value FP , meaning that multiple threads of execution can share values through a common frame. In the PIM Lite-0 chip implementation, a hardware scheduling mechanism ensures that two continuations with a common FP will never be in the pipeline at the same time, eliminat-

ing the need for hazard detection or forwarding logic in the pipeline. The simulations in this work support a more complex continuation model, including local stack data. In this work, all synchronization operations occur through memory (as each thread’s local registers cannot be shared), but the fundamental continuation and scheduling mechanisms are the same.

It should be noted that the on-chip memory is not the only memory potentially available to a PIM. A configuration in which PIMs are backed by DRAM offers several advantages: the amount of local state available to each node increases (though the latency to the off-chip memory is relatively high); the total memory bandwidth of the system is tremendously increased (given that each PIM node can access local DRAM and can serve as a bandwidth multiplier for the remaining nodes in the system accessing that data); and the memory hierarchy supports an intermediate access (as local off-chip memory will be closer than remote memory). The use of local off-chip memory does not detract from the inherent advantages of PIM: large, low latency, high bandwidth local memory.

2.4. Multithreading

The PIM execution model used throughout this work is, fundamentally, multithreaded. To maximize the amount of on-chip area available for DRAM, the processing logic is intentionally kept very simple (relatively short pipelines, in order execution, and no branch prediction) and because the DRAM access time is relatively quick – $O(10ns)$ for a random address access, or a single clock cycle for addresses already in the DRAM’s open row buffer – there is no need for local caching. Multithreading is used to tolerate these local latencies. However, PIM threads are significantly less heavyweight than those seen in modern SMTs (such as multithreaded versions of the PowerPC[4], or hyperthreaded Intel processors[24]) or even more heavily multithreaded architectures such as the Cray(/Tera) MTA[1].

The execution model supports a wide spectrum of threads, including:

1. **Threadlets:** which are very tiny operations requiring extremely small state to represent (on the order of a cache line in a modern processor). For example: `if(condition[i]) counter[i]++;` could produce a small thread to be sent to the PIM that holds `counter[i]` requiring that it be incremented.
2. **Dispatched Threads:** representing more significant computations (which potentially require more state), such as scatter/gather operations.
3. **Traditional RPCs or Remote Method Invocations:** which represent a request for a remote object to perform an operation (via proxy).
4. **Heavyweight Threads:** which represent larger, more traditional threads, such as an iteration of an SPMD program loop.

Exploiting this mix of threads is particularly important to the library writer because these are the lightest weight mechanisms for performing communication on the given PIM architecture.

The architectural support for multi-threading is provided via a simple hardware mechanism, the *thread pool*, which holds the local thread state (very much like a register file), and allows the hardware to schedule from among the threads in the pool, potentially issuing an instruction from a different thread every clock cycle to avoid data dependencies. In its simplest form, the scheduler simply provides round-robin execution of each of the threads in the thread pool.

Synchronization is also very fine grain, light weight, and provided by hardware via the use of a Full/Empty bit (similar to that employed by the Cray MTA[1]). When using synchronizing loads and stores, a LOAD instruction will check to see if the full/empty bit for a given wide word is set to FULL, and, if so, will atomically load the data into a register and set the bit to EMPTY. If the bit is already EMPTY, the thread can either block or spin until the bit becomes FULL. STORE instructions return the data to memory and set the full/empty bit to FULL.

2.5. PIM System Architecture

Figure 2 shows the three possible PIM system architectures. In the first configuration, and throughout this work, PIMs can construct homogeneous arrays in which the PIM itself constitutes the only processing element in the system. PIMs may also be used as the memory for conventional machines, providing acceleration for local computations (as in the DIVA[18] architecture), or as part of a very large memory hierarchy (as in the HTMT architecture[20]). In any case, the homogeneous array model allows for fast PIM-to-PIM communication using MPI.

3. MPI Implementation using Parcels

The goal of MPI for PIM is to provide a viable “proof of concept” and a testbed for exploring the issues of implementing MPI on a PIM system. Specifically, it explores the effects of a highly multithreaded programming model on MPI’s complexity and performance. As a limited testbed, and due to the constraints of the Architectural Simulator, MPI for PIM implements only a subset of the MPI-1.2 standard[25]. With the exception of `MPI_Barrier()`,

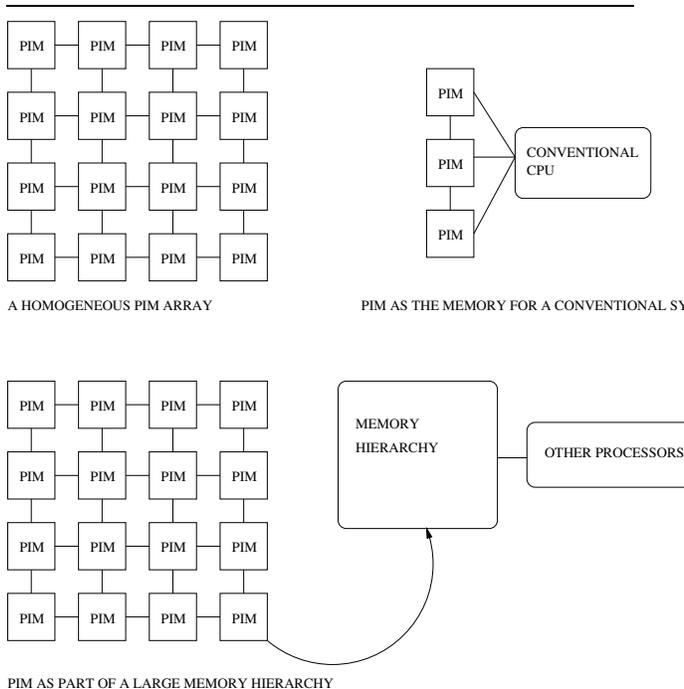


Figure 2: PIM System Architecture

MPI_Comm_rank()	MPI_Isend()
MPI_Comm_size()	MPI_Recv() †
MPI_Finalize()	MPI_Send() †
MPI_Init() †	MPI_Test()
MPI_Irecv()	MPI_Wait() †
MPI_Probe()	MPI_WaitAll() †
MPI_Barrier() †	

Figure 3: Subset of MPI implemented by MPI for PIM. † indicates functions which are built from other MPI functions.

only basic point-to-point communication and basic support functions were implemented (Figure 3). Only support for basic MPI Datatypes is included and `MPI_COMM_WORLD`, is the only group. Also, each MPI process runs in the same address space. Due to the highly multithreaded implementation of MPI for PIM, many of the blocking communication functions are built from their equivalent nonblocking functions and an `MPI_Wait()`.

3.1. Effects of threading

MPI for PIM uses pervasive multithreading to achieve concurrency, reduce the complexity of the implementation, and hide latency. To avoid the traditionally high

costs of thread synchronization and programming, MPI for PIM leverages two features of the PIM programming model: fine-grain locking and thread migration.

Conventional single thread implementations of MPI often have difficulty achieving true concurrency with non-blocking communication. After requests are enqueued, the status of the request can only be advanced when a call is made to MPI. Thus, whenever any MPI call is made, a single thread MPI must iterate through its list of outstanding requests and attempt to update their status. This results in many MPI calls experiencing significant overhead when there are outstanding requests as the MPI implementation must “juggle” all outstanding requests whenever an MPI call is made [38]. By using threads, MPI for PIM avoids juggling requests. Requests are assigned a thread which can advance the request as needed without having to wait for another call to MPI.

Because PIMs utilize fine-grain interwoven threads, it is possible to use threads to reduce or hide latency. For example, a call to `memcpy()` in a single threaded MPI implementation would cause all other processing to stop for the duration of the call. In contrast, MPI for PIM can divide a `memcpy()` amongst several threads allowing the copy to proceed in parallel with other processing. By using multiple threads for each `memcpy()`, it is possible to fully utilize the processor pipeline by avoiding stalls.

Traditionally, synchronization between threads has carried a high cost. Conventional mutexes and semaphores require expensive context switches between user and OS space and incur significant overhead. MPI for PIM avoids much of this cost by using hardware supported fine-grain locking through full-empty bits (FEBs). FEBs associate an extra bit with each wide word (256 bits) of memory. This bit can be used to provide mutual exclusion of accesses to that wide word. For example, if a thread accesses a word with a full-empty bit set to empty (0) that thread will block. A unique identifier for the blocking thread is stored so that when another thread ‘fills’ that FEB (setting it to 1), the blocking thread can be quickly woken. Because of this hardware support, FEBs can be utilized without a context switch to the OS, and with very little performance overhead. This low overhead also allows use of locking at a lower level of granularity, limiting serialization and again improving performance.

Another key aspect of the PIM programming model is support for traveling threads. Traveling threads allow the communication of not just “dumb” data, but also a thread of execution. In MPI for PIM, this means that a receiving process does not have to dedicate resources to monitoring incoming messages and responding to them. Instead a message send causes a thread migration to the destination process. Once there, the sending thread continues execution, performing any required resource management or copying

as needed. Because each incoming message is a thread, it can “look after itself.” This again avoids having to juggle multiple MPI requests.

3.2. Key Data Structures

Each MPI process has three main queues which coordinate communication between the threads on that node:

- **Posted Queue:** contains MPI requests for receive operations which have posted a buffer to be received into, but which are not yet completed. Calls to `MPI_Irecv()` add to this list.
- **Unexpected Queue:** contains requests from messages which arrived at an MPI process, but could not find a posted buffer to be copied into. These messages will allocate a buffer and copy their data to it.
- **Loitering Queue:** used for the rendezvous send protocol. Large messages which arrive unexpectedly may not be able to allocate sufficient resources to create an unexpected buffer. These messages can choose to “loiter,” periodically checking the posted queue for a suitable buffer. Loitering messages will post an MPI envelope to the loitering queue so that calls to `MPI_Probe()` will be able to match their envelope.

Each of these queues is implemented as a collection of pointers, with each of these pointers protected by a full empty bit. This allows multiple threads to traverse the queue at the same time, though only one thread can modify a particular queue element at any one time.

3.3. Implementation of `MPI_Isend()`

All calls to `MPI_Isend()` cause a new thread to be spawned. This thread will take one of two different paths of execution, depending on the message size. Figure 4 illustrates the implementation of `MPI_Isend()` in MPI for PIM. Dashed lines are used to show the flow of the calling thread with solid lines illustrating the flow of the `Isend` thread.

Data buffers for “Eager” messages (below 64K) are immediately assembled into a parcel for transfer across the network. Once assembled, the `MPI_Isend()` request can be marked as “done” and the thread will migrate to the destination process. Upon arriving, the `Isend` thread checks the posted queue to see if a buffer has already been posted for it. If it finds a match it will deliver the message data to that buffer. If no match is found, the thread will allocate a suitable buffer and place a request on the unexpected queue.

Messages larger than 64K utilize a rendezvous protocol to avoid resource exhaustion on the destination node. The

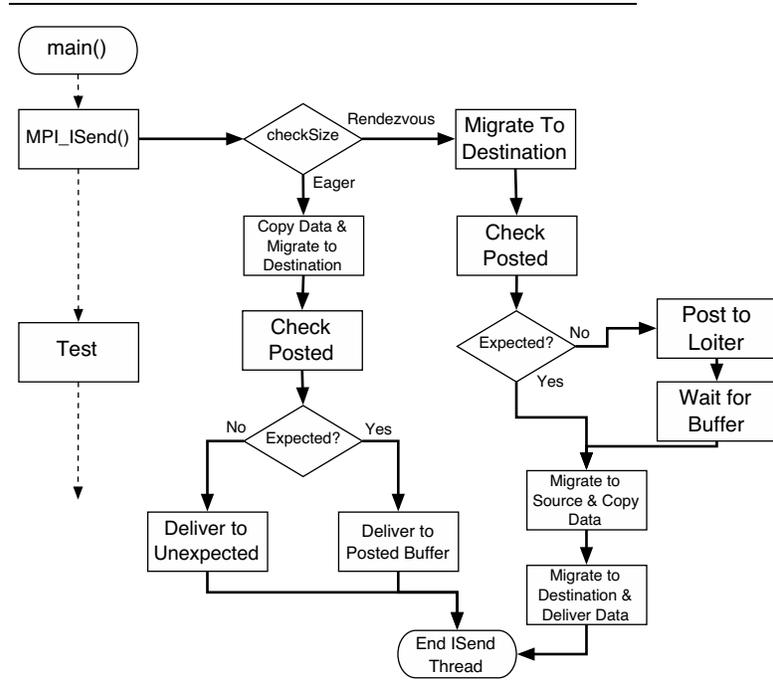


Figure 4: Implementation of `MPI_Isend()` in MPI for PIM

`Isend` thread migrates to the destination node and checks for a posted buffer. If it finds such a buffer the thread will claim the buffer and prevent other threads copying data into it by removing it from the posted queue. The `Isend` thread will then return to its source node and assemble the message buffer for transfer across the network, marking the send request as done before migrating to back to the destination node. Here the thread will deliver the message data to the waiting buffer.

If a rendezvous `Isend` cannot find a posted buffer it can instead post its message envelope to the loiter queue and then wait for a buffer to become available. By posting its envelope to the loiter queue, calls to `MPI_Probe()` can be aware of loitering rendezvous messages. To preserve ordering semantics, a “dummy” request is placed in the unexpected queue.

3.4. Implementation of `MPI_Irecv()` and `MPI_Probe()`

`MPI_Irecv()` and `MPI_Probe()` both follow somewhat similar paths (Figure 5). Because `MPI_Irecv()` is nonblocking, it begins with a thread spawn. `MPI_Probe()` is blocking, so it does not execute in another thread.

`MPI_Irecv()` first checks the status of its request, as it may already have been completed by a send. If it is not

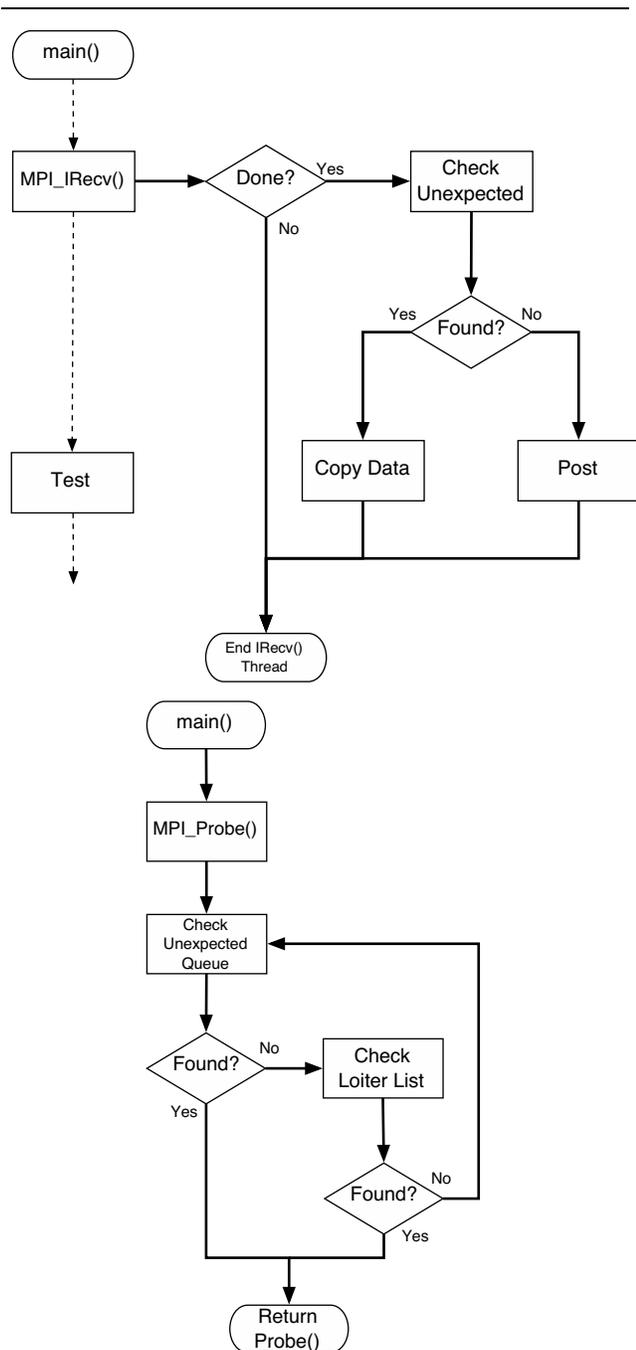


Figure 5: Implementation of `MPI_Irecv()` and `MPI_Probe()` in MPI for PIM

complete, the `Irecv` thread will check the unexpected queue for a match. If no match is found, it will post its request to the `posted` queue before exiting. It is possible for a matching send to arrive after the unexpected queue has been checked, but before the receive has been posted. This could violate the MPI ordering semantics, so the unexpected queue is locked while it is being checked and the receive is posted.

Similarly, `MPI_Probe()` also checks the unexpected queue for a match. If no unexpected message is found, it will then check the `loiter` queue to see if there are any matching rendezvous sends. `MPI_Probe()` will continue checking these queues until a match is found.

4. Evaluation Methods

The initial comparison of MPI for PIMs and commodity processors is focused on measurements of the complexity of the code paths for some core MPI routines. Thus, it is based on a simplistic microbenchmark. Traces of this microbenchmark under a variety of possible usage scenarios were taken and compared for MPICH 1.2.5 and LAM-MPI 6.5.9 on a PowerPC and for MPI for PIM on a simulated PIM architecture. This section describes the benchmark and the methodology that was used for tracing and simulation.

4.1. Benchmark

The microbenchmark used for this evaluation was written at Sandia National Labs to consider the impact of posted versus unexpected receives. The code uses a combination of `MPI_Irecv`, `MPI_Send`, `MPI_Recv`, `MPI_Barrier`, `MPI_Probe`, and `MPI_Waitall` to control the percentage of messages that are unexpected. The test sends 10 messages of parameterizable size in each direction (for a total of 20 sequential sends). This benchmark was used for this analysis because it effectively exercised a small set of the most commonly used MPI routines under varying usage scenarios. This allowed us to vary the code paths taken and study the impact of those code paths on instruction count, memory references, and instructions per cycle (IPC).

4.2. Trace Based Analysis

Traces for the baseline conventional implementations LAM and MPICH were gathered on an Apple Macintosh Power Mac with a PowerPC MPC7450 (G4+) processor running at 1Ghz. This platform was running Darwin kernel version 6.6 (Mac OS X 10.2.6). The `amber` utility [2] was used to gather instruction traces of the microbenchmark described in Section 4.1 using both LAM and MPICH implementations of MPI. These instruction traces were then

converted to an architecture independent format called TT7 [35] for further analysis.

The traces gathered from LAM and MPICH were simulated on a PowerPC MPC7400 (G4) processor running at 400Mhz to perform more detailed analysis. The MPC7400 can fetch up to four instructions per cycle and can have up to eight instructions executing at a time. Microarchitecturally, it has two integer units, one combined address generation and memory unit, a floating point unit, and branch resolution unit, and two vector units. The PowerPC has a 32K 8-way associative iL1 and dL1 and a 1024K 2-way combined L2 cache [26]. For these simulations the caches and TLBs were warmed. Though the platform used for tracing and the platform used for simulation differ, they both utilize the same PowerPC ISA, so the instruction traces are portable.

Execution of MPI for PIM was performed on a PIM Architectural simulator which can also generate traces. The MPI for PIM source code was instrumented with special tracing functions so instructions in the trace could be categorized into broad categories (see Section 5.2). To generate execution times for MPI for PIM, the traces from the architectural simulator were simulated on a PIM Trace-based simulator, built to mesh with the architectural simulator. This simulator models several key components of a PIM system and uses a discrete event simulator to represent interactions between these components. The simulator uses the instruction trace of the execution of a program to model the behavior and execution of that program on a hypothetical PIM system. A number of architectural parameters for this hypothetical system can be specified for the execution of the trace. These parameters include: the manner in which data is distributed amongst the PIMs, memory latencies, communication latencies, PIM memory sizes, instruction cache parameters, and pipeline depth.

To provide a fair comparison between MPI for PIM and other implementations, sections of the LAM and MPICH traces which concerned functionality not implemented in MPI for PIM were discounted. These include functions which dealt with specifics of the network interface, book-keeping, debugging, datatype or communicator lookup, byte ordering, and parameter checking. Such functions were identified and any instructions in the trace which executed in these functions were removed. To accomplish this, the `otool` disassembler was used to find mappings between instructions in the TT7 Trace and functions in LAM or MPICH.

4.3. Simulation Based Analysis

Cycle counts for execution on the PowerPC were obtained using the `simg4` cycle accurate simulator from Motorola [27]. This simulator produced accurate cycle counts,

instruction mixes, pipeline stall counts, and cache performance data. Cycle count estimates for the instruction categories for each function shown in Section 5.2 were estimated using output from `simg4`. Pipeline stall counts for memory instructions were used to calculate a rough IPC for memory instructions. Given this number, the number of memory instructions, and the overall number of cycles to execute the function trace, it was possible to estimate the average IPC of non-memory instructions for that function. The relative number of memory to non-memory instructions belonging to each instruction category were combined with the IPC estimates to produce a cycle estimate for each category.

The PIM Architectural simulator is a component-based discrete event simulator. It is based off of the SimpleScalar tool set [9] and uses the PISA ISA with special extensions to access extra PIM functionality such as thread migration, thread creation, and the manipulation of Full/Empty Bits. These extensions are consistent with the PIM Lite ISA. It can simulate the functioning of multiple PIMs and includes support for adjusting some architectural features, such as the distribution of the address space across multiple PIMs and network latency. Variables for memory access, pipeline delays, and instruction fetching can all be adjusted (table 1).

5. MPI Performance Impact

This section presents results comparing various aspects of the performance of the MPI for PIM prototype and MPI implementations on commodity platforms. As described in Section 4, only the aspects of MPI that were implemented in MPI for PIM were analyzed. Comparisons are presented for both eagerly sent messages (256 bytes) and messages transferred with a rendezvous protocol (80 KB). The comparisons include the number of instructions executed, the number of memory accesses performed, the instructions per cycle (IPC) achieved and the total number of CPU cycles. These numbers are presented on an overall (Section 5.1) and a per MPI call basis (Section 5.2). Some discussion of other performance issues is also presented in Section 5.3.

5.1. Overhead Reduction

An important aspect of MPI for PIM is the potential reduction in the overhead of MPI calls. MPI overhead includes time spent performing tasks other than the actual network communication or required buffer copies. Because of a pervasively multithreaded implementation, MPI for PIM can avoid much of the MPI state swapping, or “juggling”, which must occur in a single thread MPI.

MPI for PIM executes fewer overhead instructions than LAM, and usually fewer instructions than MPICH, depending on message size and the number of posted receives

Variable	simg4	PIM
Main memory latency, open page	20 cycles	4 cycles
Main memory latency, closed page	44 cycles	11 cycles
L2 latency	6 cycles	NA
Pipelines	7 (2 int., mem, FP, BR, 1 Vec.)	1
Pipeline Depth	4 (integer)	4 (interwoven)

Table 1: Latencies and processor configurations used for simulation

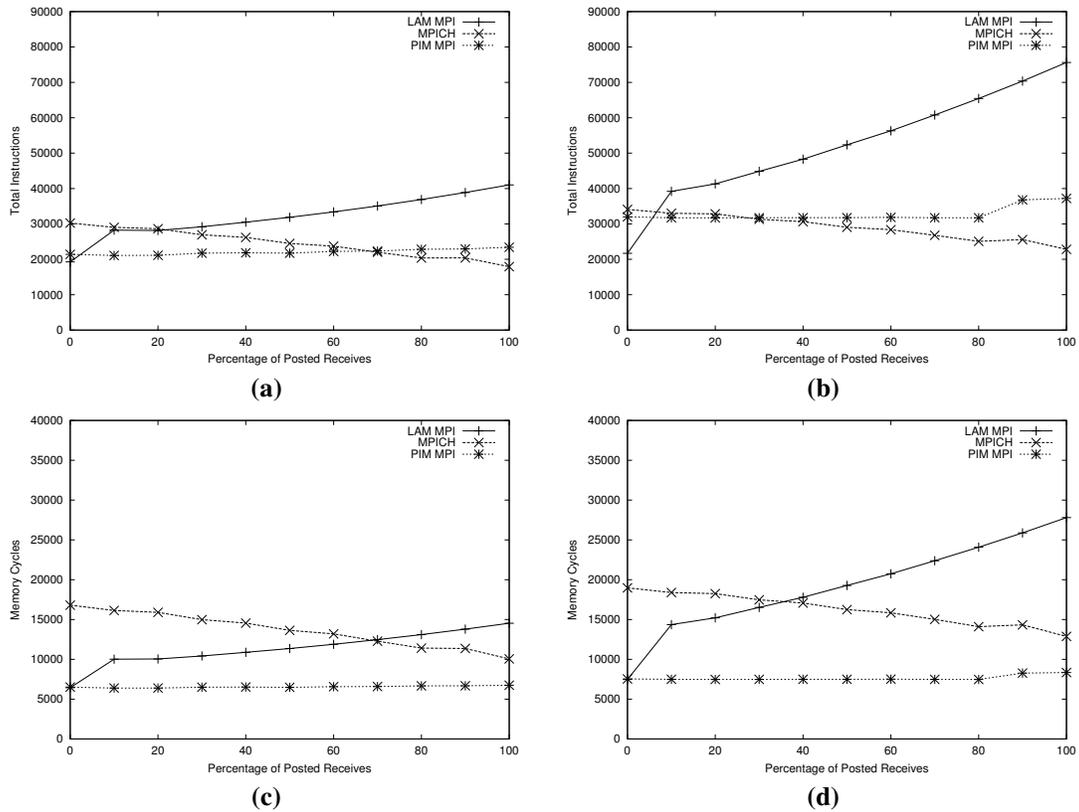


Figure 6: Total instructions executed in MPI routines for benchmark application (a) eager sends and (b) rendezvous sends, excluding network instructions; Number of memory accesses in MPI routines for benchmark application for (c) eager sends; (d) rendezvous sends, excluding network instructions.

(figure 6(a-b)). The PIM implementation also makes fewer memory references (figure 6(c-d)). The reduction in memory references is compounded because the PIM processor is “closer” to the memory. As such, these memory references tend to be less costly (lower latency, higher bandwidth) than memory references in a conventional architecture. Combining the reduction in memory references with the improvement in memory access time yields a significant reduction in the time spent accessing memory.

Because MPI for PIM’s memory references are fewer

and faster, its overall IPC tends to be high (figure 7(c-d)). MPICH suffers from a high branch misprediction rate (up to 20%), which usually limits its IPC to less than 0.6. LAM’s IPC for eager messages is high, often outperforming PIM. However, for longer messages it suffers from more data cache misses which limit its performance.

These differences in IPC result in an overall cycle count which is lower than the conventional MPIs. For eager sends, MPI for PIM averages 45% less overhead than MPICH and 26% less than LAM. For rendezvous sends, MPI for

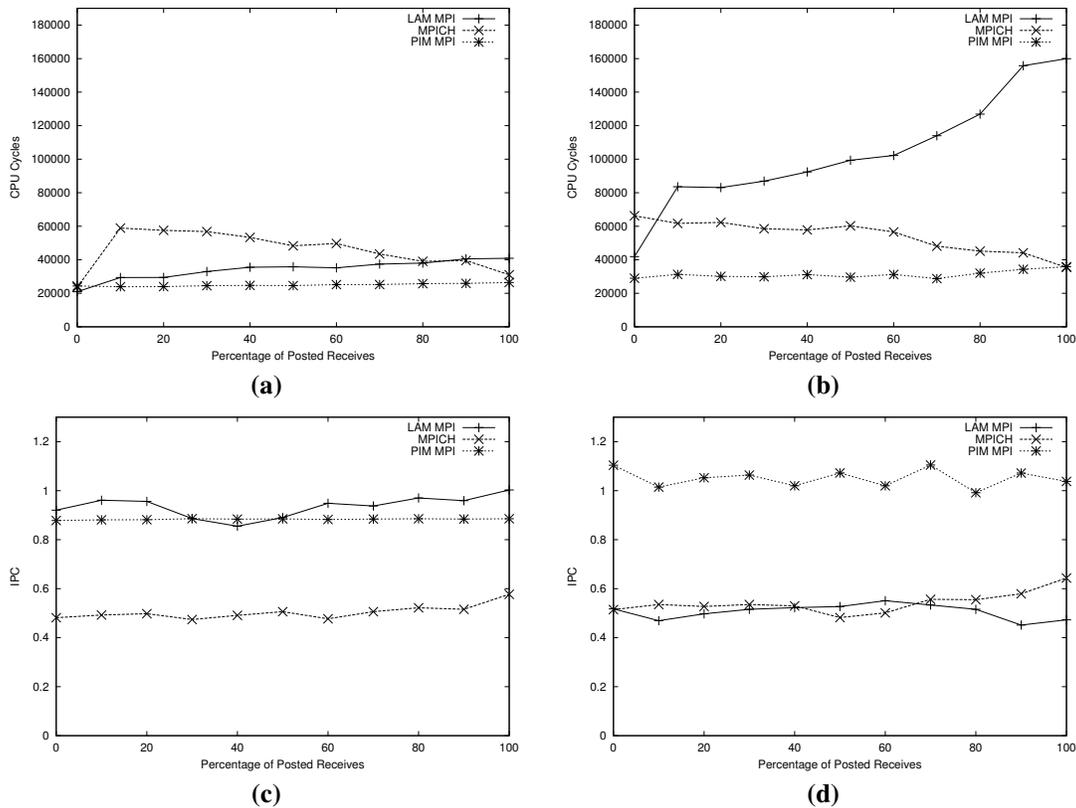


Figure 7: Total CPU cycles spent in MPI routines for benchmark application for (a) eager sends and (b) rendezvous sends, excluding network instructions; Instructions per cycles (IPC) for instructions in MPI routines for benchmark application for (c) eager sends and (d) rendezvous sends;

PIM averages 42% less overhead than MPICH and 70% less than LAM.

The actual time spent in MPI would depend on the fabrication process used in a PIM processor. However, a PIM pipeline would generally be much simpler than a conventional processor and would probably be able to run at least a similar clock rate. Additionally, as conventional processor speeds grow, the latency between memory and processor would also increase further limiting conventional performance.

5.2. MPI Function Analysis

To explain the performance differences between MPI for PIM and conventional single threaded MPIs, it is useful to examine several of the major MPI calls. The overhead in these calls can be classified into one of four behaviors:

- **State Setup/Update:** Initialization and updating of MPI Requests and internal state dealing with the progress of a function.

- **Cleanup:** Deallocation of data structures, unlocking of synchronization controls, removal of requests from lists or queues.
- **Queue Handling:** Iterating through lists or queues to advance requests or match envelopes. May also include searching hash tables for matches (LAM) and acquiring synchronization locks (MPI for PIM).
- **Juggling:** Time spent switching from the MPI context of one request to another in single threaded MPIs. This generally occurs when there are multiple outstanding non-blocking requests and the MPI must check each to see if progress can be made on them, such as LAM's `rpi_c2c_advance()` or MPICH's `MPID_DeviceCheck()`.

MPI for PIM generally executes MPI functions with less overhead than single threaded MPIs. This difference in performance comes from several factors. In addition to the overall performance improvement of faster memory accesses, MPI for PIM functions require less state setup for

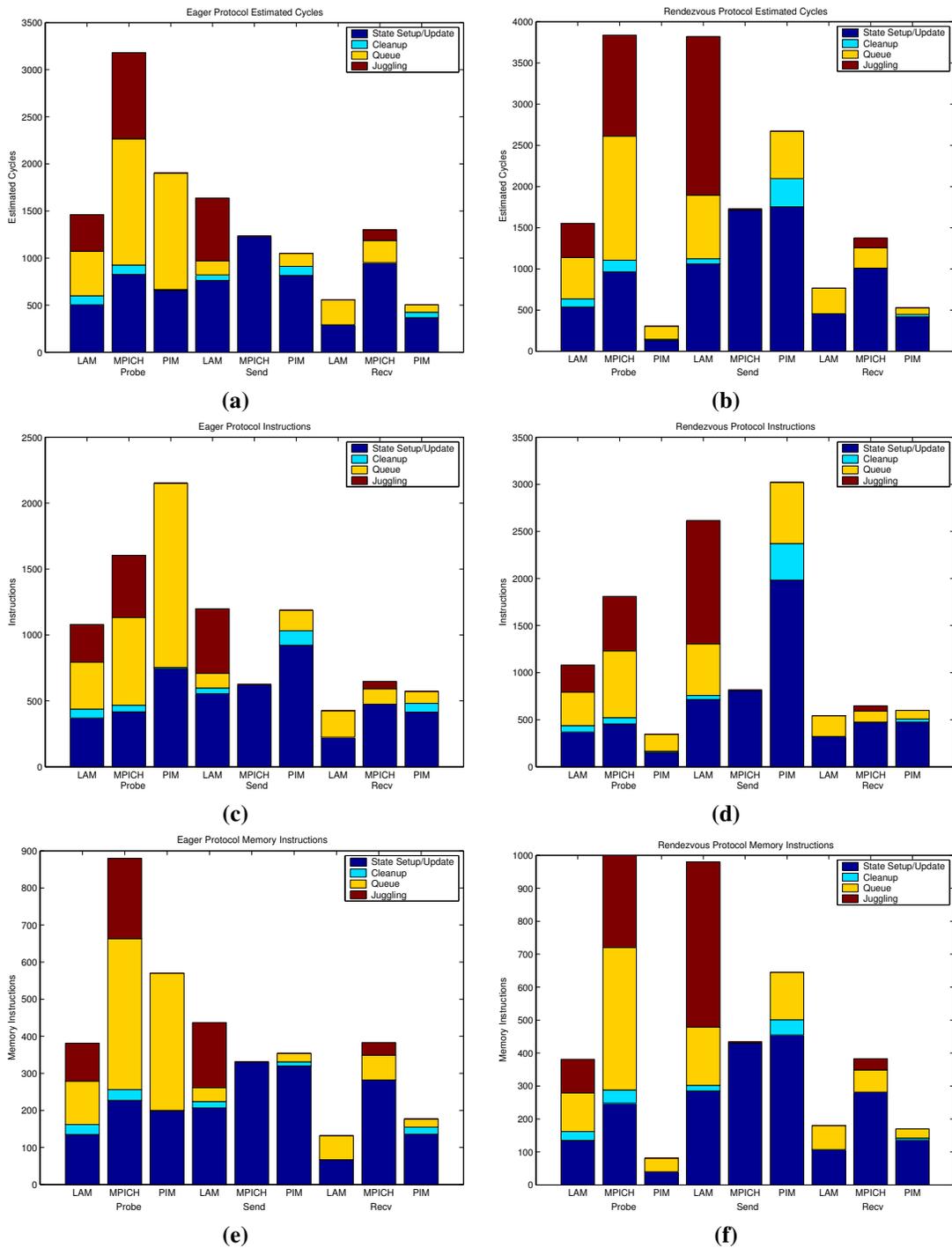


Figure 8: A breakdown of the CPU cycles spent in each of three routines for (a) eager sends and (b) rendezvous sends; A breakdown of the instructions executed in each of three routines for (c) eager sends and (d) rendezvous sends; A breakdown of the memory access instructions executed in each of three routines for (e) eager sends and (f) rendezvous sends. All breakdowns exclude network and memory copy instructions.

the rendezvous protocol, and does not have to “juggle” multiple requests.

MPI for PIM requires fewer cycles to setup and maintain state in several key MPI functions, but especially when comparing the rendezvous protocol (figure 8(a-b)). This is due to the use of “intelligent” traveling threads to perform sends. A conventional MPI must expend cycles initializing a send request, update this request as it is sent, and then interpret the incoming data, dispatch it based upon protocol, and setup state on the receiving side to track the incoming data. In effect, a conventional MPI must setup the state information for send twice. In contrast, an MPI based upon traveling threads does not have to interpret and dispatch incoming data. Instead, the incoming thread contains state describing the send which is already initialized. The traveling thread can “dispatch itself”, performing whatever actions are required by its protocol.

Another advantage of MPI for PIM is that MPI functions do not have to switch contexts from one MPI request to another to advance pending requests. The overhead of this “juggling” of requests can be quite significant (figure 8(c-d)), especially since this class of behavior tends to require a large number of memory accesses (figure 8(e-f)). In LAM it accounted for 14% to 60% of MPI overhead instructions, depending on the number of outstanding requests. In MPICH, it accounted for between 18% and 23%.

There are some cases where MPI for PIM performs poorly compared to LAM or MPICH. LAM’s implementation of `MPI_Probe()` outperforms MPI for PIM, mainly due to inefficient queue traversal in MPI for PIM. This is most likely because MPI for PIM’s `MPI_Probe()` must cycle between two queues. Additionally, MPICH’s `MPI_Send()` outperforms MPI for PIM with rendezvous sized messages. It appears that MPICH’s send performs a “short-circuit” type optimization and bypasses the normal queuing and device checking procedures. Lastly, MPI for PIM often requires more instructions in cleanup activities. This is mainly due to the extra queue unlocking which is required for synchronization.

5.3. Other Performance Impacts

Another architectural advantage of PIMs is extremely high memory bandwidth, which could be exploited for performing very fast memory copies. Conventional processors suffer significant performance degradation when performing memory copies which exhaust their cache. This effect is shown in figure 9(d). A PowerPC G4 with a 32K L1 data cache is capable of performing a `memcpy()` of less than 32K at an IPC close to 1.0. However, memory copies greater than 32K show a serious drop in performance, with IPC falling to under 0.4. This drop in performance is a graphic depiction of hitting the “memory wall” and will only be-

come more pronounced as the gap between memory and processor speeds grows.

PIM processors have several advantages when performing memory copies. The first is that a PIM processor is “closer” to memory. It does not have to go through several layers of cache, but is connected directly to the memory macro. Additionally, it is possible to copy a full DRAM row at a time.

MPI frequently requires memory copies to handle unexpected messages, pack data, assemble messages, and perform shared memory communication. These memory copies can account for a significant percentage of the total time spent in MPI, especially for large message sends. By utilizing the architectural features of PIM to reduce memory copy times, MPI time could be considerably reduced (figure 9(b) and (c)).

6. Related Work

The prototype MPI implementation that we have described in this paper is very similar to other MPI implementations on top of active message layers, such as those described in [12, 22, 3]. It is also similar to implementations that have been built on networks that have remote DMA (RDMA) capability, such as those described in [5, 16, 23]. However, the traveling thread model is able to support some features of MPI much more efficiently.

For example, most of the implementations of MPI on top of active messages require the process to poll the network in order to process messages and activate message handlers. This can lead to inefficiencies when the receiving process is not running, and, in some cases, may violate the progress rule of MPI. Hardware support for traveling threads increases the ability of remote processing to occur on the arrival of messages without interference from the operating system and without requiring the receiving process to waste processor cycles polling the network.

Existing RDMA-based implementations of MPI also suffer from similar issues. Messages can arrive without explicitly polling by the receiver, but the MPI library must actively notice incoming messages and process them. For example, a short message is typically written into a buffer that is managed by the MPI library, and is later copied into a receive buffer. This can only occur after the MPI library notices that it has arrived. Traveling threads allow for this processing to happen immediately upon thread arrival.

7. Conclusions

This work is based on a PIM architecture that could readily form the basis for commodity cluster computing in the future. As such, it is important to consider the implications of this technology for current computing paradigms. Thus,

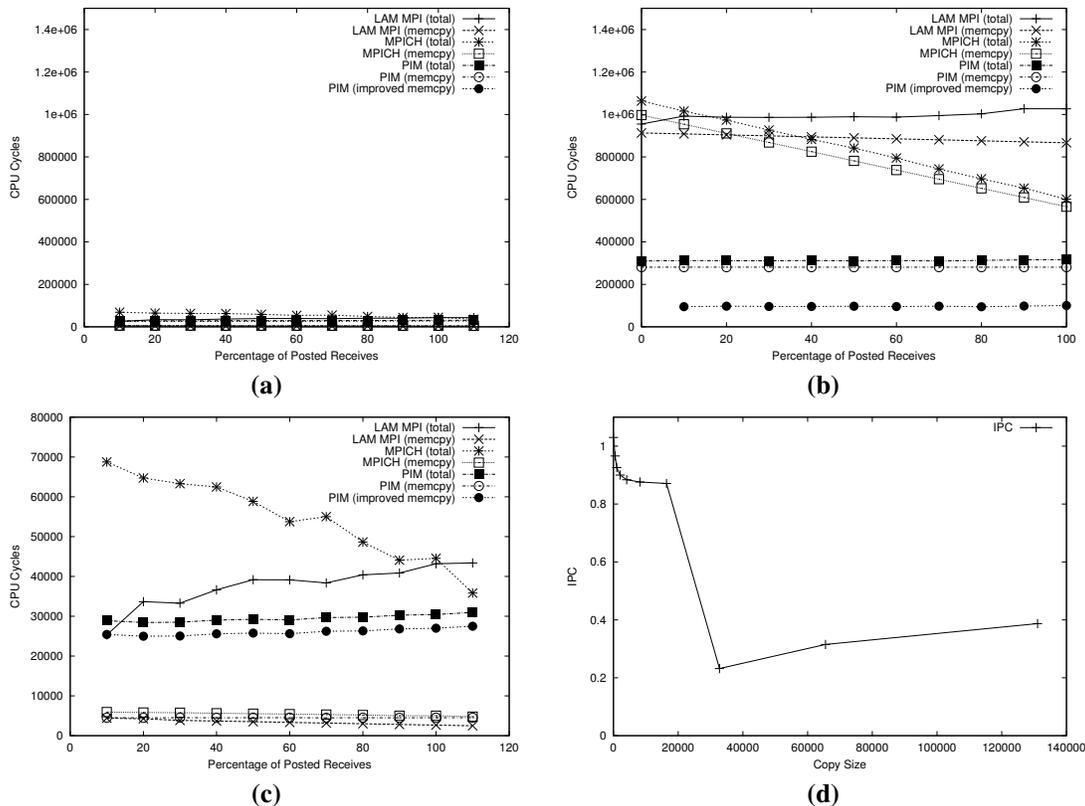


Figure 9: Total MPI cycles, including memcpys for (a) eager sends and (b) rendezvous sends; (c) eager sends at a more detailed scale; (d) Conventional memcopy IPC for varying copy sizes.

this work presents an analysis of an initial implementation of MPI on PIM architectures. Although the PIM architecture is explicitly designed for parallelism, it is not explicitly designed to support MPI. Despite this, the preliminary analysis indicates that a PIM architecture will support MPI very well, and may reduce the complexity of the MPI implementation via inherent multithreading. In terms of performance, for many of the operations implemented, MPI for PIM requires fewer CPU cycles than the equivalent implementations on a commodity processor. This is attributable to a significant reduction in total instructions through the use of special features in the PIM. In other cases, this comes from an increase in instructions per cycle (IPC). Overall, this work demonstrates that an MPI implementation for PIM is not only possible, but is likely to perform at least as well as what is found on commodity systems.

8. Future Work

Only a preliminary analysis is presented here. Future work will focus on implementing more of the MPI standard to permit application simulation on the architectural sim-

ulator. Simulation of real applications will allow us to explore PIM usage models ranging from one PIM “node” per MPI rank to several PIM “nodes” per MPI rank. This will offer insight into the balance between fine-grained parallelism extracted by a compiler (or represented in OpenMP) and coarse grained explicit message passing implemented by the user. Balance factor issues such as “surface to volume” ratios will come into play in these studies.

A second aspect of future work will be determining the impacts of other unique features of the PIM on MPI performance. For example, PIM instruction sets will likely provide vector types of operations on extremely wide words. Additionally, the extremely high memory bandwidth provided by PIMs may offer a significant win for applications using MPI derived datatypes. Also, PIMs can offer extremely fine grained synchronization methods that will allow automated exploitation of opportunities for communication and computation overlap. For example, it may be possible to allow an `MPI_Recv` to return before all of the data has arrived. Fine grained synchronization could then block the application if it attempted to access a portion of the data that has not arrived. Finally, PIMs may also sup-

port the MPI-2 one-sided communication functions very efficiently, especially the accumulate operation, which allows for operations to be performed on remote data. Finally, given the close ties between the PIMs simulated in this work and PIM Lite, experimentation with the MPI library in the PIM Lite architecture and simulations using larger MPI-based applications will be performed.

References

- [1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera System. *Tera Computer Company*.
- [2] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X*. Apple Computer Inc, July 2002.
- [3] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1081–1093, Oct. 2001.
- [4] J. Borkenhagen, R. Eickemeyer, R. Kalla, and S. Kunkel. A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research and Development*, 44(6), November 2000.
- [5] R. Brightwell and A. Skjellum. MPICH on the T3D: A case study of high performance message passing. In IEEE, editor, *Proceedings. Second MPI Developer's Conference: Notre Dame, IN, USA, 1–2 July 1996*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.
- [6] J. Brockman, P. Kogge, S. Thoziyoor, and E. Kang. Pim lite: On the road towards relentless multi-threading in massively parallel systems. Technical Report TR-03-01, Computer Science and Engineering Department, University of Notre Dame, 384 Fitzpatrick Hall, Notre Dame IN 46545, February 2003.
- [7] J. B. Brockman, P. M. Kogge, V. Freeh, S. K. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ICS*, 1999.
- [8] D. Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.
- [9] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.
- [10] D. Burger, J. R. Goodman, and A. Kagi. Limited Bandwidth to Affect Processor Design. *IEEE Micro*, November/December 1997.
- [11] D. Burger and A. Kagi. Memory bandwidth Limitations of Future Microprocessors. *Proceedings of the 23th International Symposium on Computer Architecture*, May, 1996.
- [12] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-latency communication on the IBM RISC System/6000 SP. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press.
- [13] D. E. Culler, S. C. Goldstein, K. E. Schauer, and T. Von Eicken. TAM – A compiler controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.
- [14] D. E. Culler, A. Sah, K. E. Schauer, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Apr. 1991. Published in Vol.26, No.4. Apr.1991. Also as Tech report UCB-CSD-90-594, University of California Berkeley, Department of Computer Science.
- [15] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler. The message-driven processor. *IEEE Micro*, pages 23–39, Apr. 1992.
- [16] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In *Proceedings of the Third MPI Developers' and Users' Conference*, pages 15–24, March 1999.
- [17] R. Draves, B. Bershad, R. Rashid, and R. Dean. Using Continuities to Implement Thread Management and Communication in Operating Systems. *Proceedings of the 13th Symposium on Operating Systems Principles*.
- [18] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.
- [19] P. M. Kogge, J. B. Brockman, and V. Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27–28, 1998.
- [20] P. M. Kogge, J. B. Brockman, and V. W. Freeh. PIM Architectures to Support Petaflops Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1–3, 1999.
- [21] S. K. Kuntz, R. C. Murphy, M. T. Niemier, J. Izaguirre, and P. M. Kogge. Petaflop Computing for Protein Folding. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA*, March 12–14, 2001.
- [22] M. Lauria and A. A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.
- [23] J. Liu, J. Wu, S. P. Kinis, P. Wyckoff, and D. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of 17th Annual ACM International Conference on Supercomputing*, June 2003.
- [24] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, February 2002.
- [25] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

- [26] Motorola Inc. *MPC7400 RISC Microprocessor User's Manual*, March.
- [27] Motorola System Performance Modeling and Simulation Group. *Sim_G4 v1.4.1 User's Guide*, 1998. Available as part of Apple Computer's CHUD tool suite.
- [28] R. C. Murphy. *Design Parameters for Distributed PIM Memory Thesis*. MS CSE Thesis, University of Notre Dame, April 2000.
- [29] R. C. Murphy and P. M. Kogge. Trading Bandwidth for Latency: Managing Continuations Through a Carpet Bag Cache. In *Proceedings of the International Workshop on Innovative Architecture 2002 (IWIA02)*. IEEE Computer Society, January 10-11, 2002.
- [30] R. C. Murphy, P. M. Kogge, and A. A. Rodrigues. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems. In *Proceedings of the Second Workshop on Intelligent Memory Systems, held in conjunction with ASPLOS-IX, Cambridge, MA*. ACM Press, November 12-15, 2000.
- [31] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, June 1989.
- [32] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine multicomputer: An architectural evaluation. In L. Bic, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–236, San Diego, CA, May 1993. IEEE Computer Society Press.
- [33] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *17th International Symposium on Computer Architecture*, number 18(2) in ACM SIGARCH Computer Architecture News, pages 82–91, Seattle, Washington, May 28–31, June 1990.
- [34] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.
- [35] A. Rodrigues. Rudra: A Reactive Dissipation Reducing Architecture. Master's thesis, University of Notre Dame, 2003.
- [36] T. Sterling and L. Bergman. A design analysis of a hybrid technology multithreaded architecture for petaflops scale computation. In *International Conference on Supercomputing, Rhodes, Greece*, June 20-25, 1999.
- [37] T. Sterling and H. Zima. Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing. In *SC2002, Baltimore, MD*.
- [38] T. L. Team. Porting the lam-mpi 6.3 communication layer. Technical Report TR00-01, Univesity of Notre Dame, 2000.
- [39] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.
- [40] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: a mechanism for integrated communication and computation. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.