Techniques for Low-Overhead and No-Overhead Communication Using Commodity Superscalar Processors

Brian Grayson and Craig Chase

TR TR-PDS-1998-015 December 1998

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Techniques for Low-Overhead and No-Overhead Communication Using Commodity Superscalar Processors
Technical Report TR-PDS-1998-015

Brian Grayson* bgrayson@ece.utexas.edu

Craig Chase chase@ece.utexas.edu

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084

December 1998

Keywords: Message-passing, network interface, superscalar, low-overhead communication, synchronization

Abstract

Previous researchers in user-level message-passing parallel computing have attempted to reduce communication overhead via several techniques: integrating part or all of the network interface (NI) on the CPU[8, 2], allowing NI device register accesses to be cached[11, 6], or using polling to avoid the overhead of interrupts[15, 18]. Unfortunately, these techniques incur significant overhead if applied to a superscalar processor system. We have developed architectural features that have the potential to alleviate these overheads. In this paper we present the results of simulations which demonstrate successful overlap of communication overhead and network latency with useful work. This benefit was achieved without sacrificing programmibility — our message-passing libraries use PVM semantics.

1 Introduction

Developments in processor design have created CPUs that are able to find and exploit low-level parallelism in sequential code through techniques such as pipelining, superscalar issue, and memory access reordering. However, many previous techniques at improving user-level message-passing on parallel computers may not "scale" to these processors. For example, interrupt-driven techniques require the processor to synchronize and save its state before the interrupt handler can be executed[15], which on superscalar processors may add dozens of cycles of overhead and disturb the dataflow execution of the processor. Many of the ideas behind Cacheable Device Registers[11] rely on the fact that memory accesses are performed in program order, but many commodity superscalar processors such as the PowerPC[10], the Alpha AXP family[17], and the Pentium Pro[7] allow memory accesses to be reordered to various degrees. Various CPU-Network Interface integration efforts such as Sparcle[8] and

*Research supported in part by a National Science Foundation Graduate Fellowship.

the Message-Driven Processor[2] of the J-Machine suffer from high development costs (effort, time, and money), which will likely increase with a re-implementation for superscalar processors, and also require significant overhead to provide support for fast context switches and fast interrupts. Lastly, polling techniques, frequently used for high performance when the cost of interrupts is too high[15, 18], prevent the processor from efficiently exploiting some available instruction-level parallelism: code following a spinloop can only be executed speculatively until the spinloop completes, even if some of those instructions do not depend on the polling condition.

While these effects are generally presumed to be small, no researchers to date have attempted to measure their impact on performance. In addition, with the use of two of the three techniques described by the authors in [5] (*synthetic dependences* and *delayed-response device registers*), interference (polling spin-loops and interrupts) with the processor's dataflow scheduling can be virtually eliminated. One key difference between our techniques and previous efforts is that our techniques can be used on commodity superscalar processors such as the PowerPC, requiring modifications only to the network interface device.

The contributions of this paper include demonstration of two techniques that allow dataflow device interactions, and a presentation of simulation results from a PVM-style implementation showing the strengths and weaknesses of these techniques.

2 Background

In [5], the authors presented three techniques to eliminate spin-loop polling and memory barriers in device routines, without requiring the use of interrupts, thus providing the compiler code that is easier to optimize, and the processor with more runtime scheduling freedom. In this paper we extend this earlier work by performing detailed simulations of several parallel benchmarks. The benchmarks are written using a library based upon PVM's message passing semantics. We are able to quantify the overhead reduction and improved latency tolerance that (*synthetic dependences* and *delayed-response device registers*) provide even to code written to a high level message passing library. For this paper, we assume that the processor allows reordering of any memory accesses if the addresses are not the same, including uncacheable accesses.

2.1 Synthetic Dependences

The most general technique is *synthetic dependences*. Generally speaking, a synthetic dependence is any introduction of a dependence between two instructions (typically through the use of register values) that would otherwise have no dependence relationship conveyed to the processor. These synthetic dependences can be used to add the proper dependence relationships between accesses to device registers, which typically would otherwise not be constrained with respect to each other. For example, if we wanted to configure a device by reading (with device side-effects) one of its device registers, and then enable some action by accessing another device register, as shown in the sample code of Figure 1, correctness may be violated if the enabling access is performed before the configuring access. As far as the processor is concerned, the two accesses are to different addresses (`0x3000` and `0x3100`), so they are free to be reordered. If synthetic dependences were not used, the programmer would need to insert a `sync` between the two accesses to prevent such a reordering.

We can delay the second device access by creating a register dependence between the load and the store. One method would be to use the return value from the first access to "modify" the address used for the second access by adding 0 to it, as shown in Figure 2. Thus, a data dependence has been added, to convey dependence information to the processor.

```
...
load val1 from 0x3000    /*  Configure device.  */
store 0 to 0x3100        /*  Enable.  */
...
```

Figure 1: Sample device driver code

```
...
load val1 from 0x3000    /*  Configure device.  */
temp = val1 & 0;         /*  Create a zero value that won't
                             be available until the read is done.  */
store 0 to 0x3100+temp   /*  Enable.  */
...
```

Figure 2: Sample device driver code with a synthetic dependence

Through the use of synthetic dependences, a `sync` instruction has been removed, while still conveying sufficient dependence information to provide correct operation.

One drawback of synthetic dependences is that the use of the effective address to create a dependence introduces an additional memory disambiguation problem. Although the example from Figure 2 could be rewritten to incorporate the synthetic dependence in the value (rather than the address) for the store instruction, synthetic dependences on load instructions must be based on the effective address — the only source operands to a load instruction are used to calculate the effective address. By doing so, the load/store unit is unable to calculate the effective address of the later access until the dependence has been satisfied, possibly causing future, non-dependent memory accesses to be delayed. Since the processor must ensure that RAW and WAW hazards are handled properly it must stall any writes until effective addresses are known for all earlier reads or writes, and it must stall any reads or writes whenever an effective address for an earlier write is unknown. Note that in any case, the use of a synthetic dependence is no more restrictive than using a memory barrier (`sync`).

2.2 Delayed-response Device Registers

The use of synthetic dependences allows the programmer to avoid the use of memory barriers in many situations. However, a similar instruction-level-parallelism impediment is the spin-loop poll of a device register. These polling loops are wasteful for several reasons:

- If the polling condition is not true (and thus multiple loops of the poll are executed), the processor is not able to accurately schedule code after the polling loop, even with branch prediction. Notice that every time the polling condition is found to be not true, any "speculative" instructions must be squashed, leading to very little useful overlapped work during the majority of the polling operation.
- From a more philosophical viewpoint, only the first and last iteration of the spin-loop provide useful information: the first iteration says whether or not the condition is true, and the last

iteration (if different from the first) says that the condition is definitely true. All the intermediate iterations provide no new information, so the resources spent on these iterations are wasted.

Although theoretically the high-level code after the polling loop depends on the results of the poll (otherwise the code could have been rearranged by the programmer), there is frequently code that does not depend on the polling condition, such as calculations of pointer addresses or loop prologues. If the control-flow dependence could be replaced with a more dataflow-friendly dependence, the code after the poll can be executed nonspeculatively and the results committed immediately. In addition, three basic blocks of code (one before the spin loop, one containing the spin loop, and one after the polling loop) can be merged into a single basic block, allowing better compile-time optimization and run-time scheduling.

A technique similar to full-empty bits can be used to replace the control-flow dependence with a data dependence. A *delayed-response device register* (DRDR) is a device register that, rather than satisfying read requests when the condition for that register is not true, blocks its reply until the condition is true. That is, the device register acts as a user-accessible read-only full-empty bit for a specific condition. When blocking, only the load instruction in the processor is "stalled," so nondependent instructions can continue to be decoded, executed, and completed.

When synthetic dependences are used with the return value from a delayed-response device register, control-flow-free device interactions can be programmed. All of the DRDRs in our virtual hardware return a value of 0, thus their value can be directly ORed (or added) with address calculations to create synthetic dependences. DRDRs in our virtual NI are cleared by an explicit device register access.

Sample code demonstrating both polling and the use of DRDRs is presented in Figure 3. In the polling version, the function repeatedly checks a status register until the status register returns a nonzero value. In the DRDR version, the processor performs a single load of the DRDR status register, and then accesses the data via a pointer that uses the DRDR return value.

```
/*  Polling version  */
...
do {
  load pollValue from 0x3000
} while !pollValue;
...
load data from 0x3100

/*  DRDR version  */
...
load drdrValue from 0x3200         /*  A DRDR is at 0x3200.  */
load data from 0x3100 + drdrValue  /*  A synthetic dependence is used
                                       to prevent the data load from
                                       being performed too early.  */
...
```

Figure 3: Sample Code For Traditional Polling and Delayed-response Device Register Methods

2.3 Related Work

Ideas similar to synthetic dependences were discussed by Pai *et al.* in [13] to hide the acquire latency in shared-memory systems. They use a technique similar to synthetic dependences, called *selective acquires*, to modify the address pointer to a locked data structure based on the outcome of an acquire attempt. This prevents the processor from loading the data structure before the lock has been acquired, while allowing nondependent instructions to execute.

Delayed-response device registers are different from traditional full-empty bits (such as those used in dataflow computers[14], the `fut` and `cfut` of the J-Machine[12], or the Cray T3E's E-registers[16]) in several ways. First of all, DRDRs are used to detect conditions, such as whether or not a message has arrived, rather than the validity of a particular word of memory. This separation of the condition from the region of memory allows more flexibility, even when DRDRs are used only to oversee accesses to memory. For example, with DRDRs, it is possible to configure a DRDR to "fire" when the contents of a received message have been examined n times (rather than "one time" as most full-empty bit methods would allow). This allows the receive buffer on the NI to be used directly by the program in a complex calculation (such as a doubly-nested loop of accesses), rather than manually copying them out to a buffer in ordinary memory. Second, DRDRs do not require modifications to the processor or the memory hierarchy. Third, the consequences of a DRDR condition becoming true are much more user-accessible: the user can specify, via synthetic dependences, what actions depend upon that condition, and which ones do not. This allows accesses to different regions of memory (typically different device registers) to be ordered in a customized way by the user, whereas full-empty bits usually operate on a single region of memory and do not provide mechanisms for chaining dependences between accesses to different regions.

3 Simulation Methodology

3.1 Virtual Architecture

This section describes many of the characteristics of our virtual architecture, as provided by the Armadillo multiprocessor simulator[4]. A diagram of a single node is presented in Figure 4. Note that the architectural parameters were motivated by the desire to model the costs of messaging on advanced dataflow processors, and were not necessarily restricted to what is "reasonable" today.

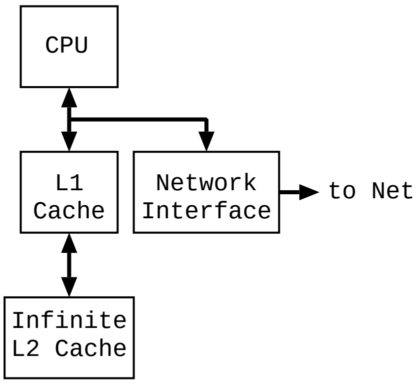

Figure 4: Single-Node Architecture

The processor is superscalar, with 4 integer pipelines, and uses the user-level instruction set of the PowerPC architecture. Most instructions execute in a single cycle. Unlimited register renaming is supported, and up to 1024 instructions can be active (decoded but not yet completed) at a given time. Each execution unit has an unlimited set of reservation stations, so instruction decode does not stall due to one unit becoming "backed up." We restrict the scope of this paper to single-process-per-node applications, although many of the techniques described here can be easily adapted for multithreaded or multiprogrammed systems. Speculative execution is not supported, so any improvement in achieving more instruction-level parallelism is from computations that are guaranteed to be useful.

The load/store unit can perform two accesses to the level 1 (L1) cache in a processor cycle. The load/store unit performs memory disambiguation, and operates under a weak consistency memory model: if two accesses are to different effective addresses, they can be performed out of order. The load/store unit also supports split transactions – if it issues an access to the L1 that can not be completed immediately, it can continue to send two new accesses per cycle to the L1 indefinitely, and can handle responses from the L1 in a different order than its requests. A `sync` instruction is available, which performs a monolithic memory barrier: no memory access after the `sync` is allowed to be performed until all memory accesses before the `sync` have completed.

The L1 access time is 1 cycle. If the L1 cache misses, it fetches from an infinite L2 with a one-line (32-byte) access time of 8 processor cycles. The L1 supports up to 8 split-transactions with the L2 at any point. For the various multiprocessor configurations, the L1 cache has a constant global size of 1MB (that is, in a 1-node configuration, there is a single 1MB L1, while in a 4-node configuration each node contains a 256KB L1, *etc.*). This avoids the artifact of speedup due to decreasing the local workload (by adding more processors) while keeping the local cache size constant.

The network interface (NI) is located at the CPU-L1 interface, although the techniques described here would also work for an NI at any point in the memory hierarchy as long as uncacheable accesses are used to access its device registers. The NI access time was chosen to be 1 cycle[1]. All of the NI's device registers are user-accessible.

The network interface contains a fixed number (currently 32) of "slots," each of which can be used to send or receive a message independent of the other slots. Each slot can hold at most a 1024-byte message, and is accessed via its own set of device registers. The sender specifies which slot should be used to receive the message, much like traditional message tags. If an incoming message is for a slot that is currently busy (either with an outgoing send or an earlier receive that has not been completed), a fixed number of such "early" incoming messages can be buffered at the receiver. Once this fixed-size buffer has been filled, all new incoming messages will be NACK'd, leading to hardware retransmits after a suitable delay.

The network model uses a fixed latency for any point-to-point message, ignoring both bandwidth and routing/contention concerns. The latency is varied from 0 to 1000 cycles for various simulations.

The compilation tool chain for the simulated executables was `gcc` version 2.7.2.2 and `binutils` version 2.7, both configured to cross-compile for the `powerpc-armadillo-elf` target architecture. All executables were compiled under `-O3` except for the `IS` benchmark, which could only be compiled under `-O2` due to a bug in `gcc`. The simulator itself is written in C++, and was compiled with `gcc` version 2.7.2.2.

[1]Researching the effects of NI device register access times on performance is an interesting but independent topic.

3.2 Message-Passing Libraries

Two libraries were developed. The first, MVP, is a PVM[3]-style communication library for our virtual hardware. Only one message can be operated upon at a time, as is typical in most PVM programs (the `pvm_setrbuf()` and `pvm_setsbuf()` functions, which remove this constraint, have not been implemented in MVP). With this library, many PVM applications can be recompiled with little modification to run on our virtual architecture.

The second library, MVP+, is loosely based on PVM, but exposes some of the features of our virtual NI to the programmer. For example, rather than manipulating a single message at a time, each MVP+ function takes a "slot" argument to tell it which of the hardware's message slots should be manipulated, in contrast to PVM's concept of having a single "active" send buffer and receive buffer. In addition, under PVM semantics, to send a message the user executes a series of `pvm_pack` calls and then enables the send via a `pvm_send` call which specifies the destination node and the tag. With our hardware, this requires MVP to copy the data from `pvm_pack` requests into a temporary buffer, and when the message is finally enabled it copies the data from this buffer into the proper slot in the NI. In MVP+, the user first calls a configuration routine which specifies the destination and the size of the message, and then proceeds to pack the message. This allows the packing to be done directly to the buffer on the NI, and also allows any overhead of the configuring to be overlapped with the packing of the data. Also, the implementation of MVP+ is cleaner than that of MVP, because less state (of the "active" buffers) needs to be kept between calls. This is reflected in the length of the code in the two libraries, as mentioned below. Sample user code for a send and a receive for the two libraries are shown in Figures 5 and 6.

```
int A[32], B[32];

/*  Send A[] to destNode.  */
mvp_pkint(A, 32);
mvp_send(destNode, destTag);

/*  Now receive into B from any node.  */
mvp_recv(-1, Tag2);
mvp_upkint(B, 32);
```

Figure 5: Sample code using the MVP library

Both of these libraries were modified to use the hardware support for DRDRs, selectable via compile-time switches, for a total of four communication libraries. The versions with DRDRs use the `sync` instruction in a few places where the use of only synthetic dependences would incur too much overhead. All of the primitives and a few higher-level functions (like barriers) are short enough to fit into a header file (less than 400 lines for MVP, less than 250 lines for MVP+), so the communication routines can be inlined, at the compiler's discretion. In addition, all of the library routines are written in C code. Unfortunately, there are a few possible optimizations that `gcc` is unable to detect, especially when synthetic dependences are involved, so hand-tuned versions of the libraries would increase performance. For this study, no hand-optimized library code was used.

None of the benchmarks were modified significantly for the MVP+ versions, even though improvement is possible, since that would distort comparisons between MVP and MVP+ versions of the same application.

```
int A[32], B[32];
int sendSlot, recvSlot;

/*  Send A[] to destNode.  */
sendSlot = mvpp_sendsetup(destNode, destTag, 32*sizeof(int));
mvpp_pkint(sendSlot, A, 32);
mvpp_send(sendSlot);

/*  Now receive into B from any node.  MVP+ does not provide
 *  selective receives based upon sending node.  */
recvSlot = mvpp_recvsetup(Tag2, 32*sizeof(int));
mvpp_brecv(recvSlot); /*  Blocking receive.  */
mvpp_upkint(recvSlot, B, 32);
```

Figure 6: Sample code using the MVP+ library

Although the use of two slightly different library styles adds yet another level of complexity to simulation results and discussion, the MVP results were kept for several reasons:

- By using a library whose semantics are common, but that is not well-matched to the underlying hardware, we show that the techniques described in this paper provide significant benefit, without requiring modifications to the applications or programming style.
- By also using a more optimized library, we can see the benefits of properly-matched library and hardware by examining the difference between MVP and MVP+ execution times.
- By using both libraries, we are able to see how much of the software/hardware mismatch overhead of MVP that is absent in MVP+ can be hidden by our techniques above and beyond the communication overheads that are hidden in both libraries.

4 Results

Some characteristics of the benchmarks used in this study are presented in Table 1. Two of the "macro" benchmarks, `SOR` and `IS`, were adapted from the PVM benchmarks in [9], while the other two macro benchmarks, `IR` and `MMULT`, and the microbenchmark `PINGN` were written by the authors. The microbenchmark is used in two ways — as a benchmark, and as a baseline for the larger benchmarks. For example, if `PINGN`, which has very little computation, saves 20 cycles per message using delayed-response device registers, and a macro benchmark saves 30 cycles per message by using delayed-response device registers, then we know that those extra 10 cycles of savings came from overlapping communication overhead with useful computation.

Note that the problem sizes are quite small for the larger benchmarks: Our research is concerned with improving the efficiency of small, fine-grain messages, so problem sizes for these benchmarks were reduced to yield message sizes of 1 to 128 32-bit words.

Each of the benchmarks will be described, followed by an analysis of our virtual architecture's communication overheads. The next two sections will then discuss the simulation results with respect to communication overhead and latency tolerance. The last section will discuss some overall conclusions from the simulation results. The average running time per iteration is presented in all of the tables and graphs.

Benchmark Name		Problem Size	Messages Sent/Received per node per iteration
Microbenchmark	PINGN	16 to 128 words	2/2
Macro benchmarks	SOR	32×32 up to 512×128	2/2
	IS	N from 256 to 4096, k from 16 to 64	1 (node P-1 sends P+1) / 2
	IR	16×16 up to 48×48 image	variable
	MMULT	8×8 global matrix size	8/8

Table 1: Benchmark Characteristics

4.1 Benchmark Descriptions

The `PINGN` microbenchmark performs a series of pings between two nodes. Each message contains N 32-bit words that are repeatedly forwarded to the other node. The first word is incremented by each node, in order to force the messages to be serialized. The `PINGN` microbenchmark contains nearly no "application computation" (a single add instruction) so the running time is determined by the network latency and the communication overhead. This benchmark allows us to identify what portion of savings in the larger benchmarks is due to better low-level communication functions (which should show up in the `PINGN` results), and how much is due to obtaining overlap of this communication overhead with useful application computation. All `PINGN` simulations were run for 100 round-trip iterations.

The `SOR` benchmark performs a red-black successive-over-relaxation calculation using an M×N matrix. The two-dimensional matrix is distributed by a 1-D block decomposition in the M dimension. In each iteration, every node sends an N-value message to its upper and lower neighbors for both the red and black matrices. The simulation was performed for several different values of M, N, total number of processors (4, 16, and 64), and net latency. The savings of the improved libraries are relatively constant across values of M (from 32 to 512) and across the number of nodes (4, 16, and 64), so the simulation results for 4 nodes are presented.

The `IS` benchmark is also from the TreadMarks study[9], and is based on the NAS IS benchmark[1]. The application sorts a large set of N numbers in a specified range of 0 to k by generating a local histogram, and then sending messages to merge all of the local histograms. Simulations were performed for three problem sizes: 256 integers from the range 0 to 15, 1024 integers from the range 0 to 31, and 4096 integers from the range 0 to 63. These problem sizes were simulated for 4, 16, and 64 nodes, with different network latencies. As stated earlier, for `IS` compilation, inlining was not possible due to a bug in `gcc`.

The `IR` benchmark performs an image rotation routine. Initially, each node contains a portion of a pixel image (distributed via two-dimensional blocking). The application then rotates the image by a certain number of degrees by having each node calculate where the new image's pixels should come from. Each node then sends its requests to the appropriate remote nodes and waits for its replies. While it waits, it handles incoming requests from other nodes. To improve performance, up to 16 pixel requests to the same node may be placed into a single message. Unlike the other benchmarks in this study, `IR` explicitly performs asynchronous communication via nonblocking receives (`pvm_nrecv()`).

Since MVP does not currently support nonblocking receives due to implementation complications, only MVP+ results for `IR` are presented.

The `MMULT` benchmark performs an 8×8 matrix multiplication on 4 nodes. The application is based on the `SUMMA`[19] algorithm: Each node performs a series of rank-1 updates, with the appropriate row- and column-vectors being broadcast within each column and row as needed.

4.2 Architectural Evaluation

Table 2 compares the communication overhead for the `PINGN` application under the various libraries. For comparison, the equivalent values for the J-Machine[12] are also included.

Library	T_s per message (in cycles)	T_w per word (in cycles)
MVP with polling	65	5
MVP with DRDRs	51	3
MVP+ with polling	20.5	4
MVP+ with DRDRs	23.5	1.9
J-Machine	11	2

Table 2: One-way Message Overhead for the `PINGN` Benchmark. T_s is the sum of the overhead for sending and receiving a message, and T_w is the cost of injection per word. All times are in cycles. Note that the J-Machine requires 2 cycles per word if the value is in a register, while MVP+ requires 1.9 cycles per word if the value is in memory (or less than 1 cycle per word if the value is in a register).

The times for T_w under MVP and MVP+ are for packing or unpacking a word from memory to a buffer on the NI, and thus require at least two instructions. If the message data can be composed from values in registers, the T_w overhead for MVP and MVP+ would be halved. Further reduction of the communication overhead is possible in real applications due to overlapping the execution of application code with this overhead, as shown in the next section.

4.3 Reduction in Communication Overhead through ILP

In real applications, the use of DRDRs and synthetic dependences allows the processor to find and exploit instruction-level parallelism near communication operations. The size of these savings can be determined by comparing the execution times of the various benchmarks when a very long net latency (1000 cycles) is used. Any overlap beyond the equivalent overlap in the `PINGN` benchmark for the same size of messages must be due to overlap of communication overhead with application code.

The results of several representative simulations are presented in Table 3. The savings column was obtained by taking the difference between execution times using polling and using DRDRs, and dividing it by the average number of messages sent or received per node per iteration. For `IS`, the number of messages per node was considered to be 32, even though the "average" number of messages sent or received per iteration per node is 8, because node 0 sends its final result to all P nodes (here, 16 nodes), which places all 16 sends (and 16 receives) on the critical path.

For `SOR`, `IS`, and `MMULT`, overlap of application code with communication overhead and communication latency has allowed the savings per message under MVP+ and DRDRs to be roughly doubled from the `PINGN` case, where no application code is available for overlap. For `IR`, the overheads of DRDRs are larger than their benefits at a net latency of 1000 cycles, as `IR` already performs useful work during the long latencies. `MMULT` is the only application whose savings from DRDRs under MVP is significantly

Application	Message size in 32-bit words	Execution time for MVP with polling	Execution time for MVP with DRDRs	Savings per message	Execution time for MVP+ with polling	Execution time for MVP+ with DRDRs	Savings per message
`PINGN` (2 procs, $N = 16$)	16	2439	2282	39.2	2297	2148	37.2
`PINGN` (2 procs, $N = 32$)	32	2758	2474	71.0	2553	2276	69.2
`PINGN` (2 procs, $N = 64$)	64	3397	2858	134.8	3064	2532	133.0
`PINGN` (2 procs, $N = 128$)	128	4675	3626	262.2	4085	3044	260.2
`SOR` (4 procs, $M = 32$, $N = 32$)	32	9200	9039	40.2	8859	8268	147.8
`SOR` (4 procs, $M = 8$, $N = 128$)	128	12884	12723	40.2	11775	10082	423.2
`IS` (16 procs, $N = 256$, $k = 16$)	16	21283	18641	82.6	20589	17565	94.5
`IS` (16 procs, $N = 1024$, $k = 32$)	32	25567	20924	145.1	24429	19179	164.1
`IS` (16 procs, $N = 4096$, $k = 64$)	64	34987	26365	269.4	32980	23298	302.6
`IR` (4 procs, $N = 16$)	16	–	–	–	33315	33463	-2.6
`IR` (4 procs, $N = 32$)	16	–	–	–	91662	91875	-1.9
`MMULT` (4 procs)	8	7215	6496	89.9	6783	6510	34.1

Table 3: Reduction in Communication Overhead through Overlap with Application Code for a Network Latency of 1000 Cycles

larger than its savings under MVP+ — only `MMULT` is able to hide a significant portion of the overhead of the MVP library.

4.4 Increase in Latency Tolerance through ILP

Besides allowing the communication overhead to be overlapped with application code, the benefit of DRDRs may be significant enough to remove network latency from the application's critical path, providing latency tolerance. For example, in Figure 8, `SOR` with polling is shown to execute in the same amount of time with a net latency of 0 cycles as with a net latency of 30 cycles – the network latency is not on the critical path unless it is longer than 30 cycles. `SOR` with MVP+ and DRDRs, however, runs in the same amount of time for any latency less than 150 cycles, for a factor of 5 improvement in latency tolerance. Figures 7 through 15 show execution time as a function of net latency for all of the benchmarks. The "knee" in each curve occurs at the point where the maximum latency tolerance is reached for that configuration.

For `PINGN`, no improvement in latency tolerance is noticed, as expected: there is very little application code to provide overlap. For `SOR`, using DRDRs under MVP provides a small amount of latency tolerance, while using DRDRs under MVP+ greatly increases latency tolerance — for $N = 128$, DRDRs provide a tenfold increase in latency tolerance. `IS` has more modest but substantial improvements in latency tolerance, while DRDRs provide little improvement for `IR`.

The graph in Figure 15 for `MMULT` shows two interesting results: MVP with DRDRs provides a small amount of latency tolerance for low latencies, shown by the decreased but nonzero slope from latencies of 0 to 50 cycles. Also, beyond 50 cycles, the extra overhead of MVP versus MVP+ is completely overlapped with DRDRs, allowing the execution times for the two versions to be nearly identical.

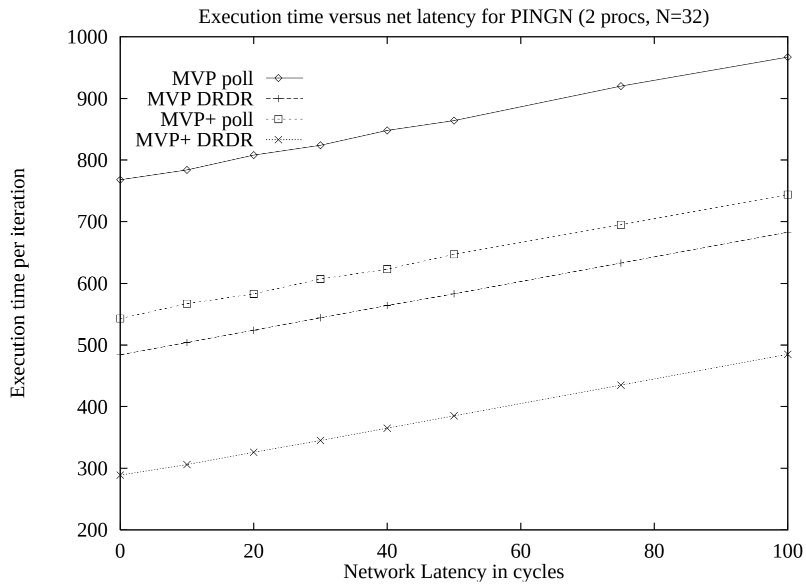

Figure 7: Execution Time versus Net Latency for `PINGN` with $N = 32$

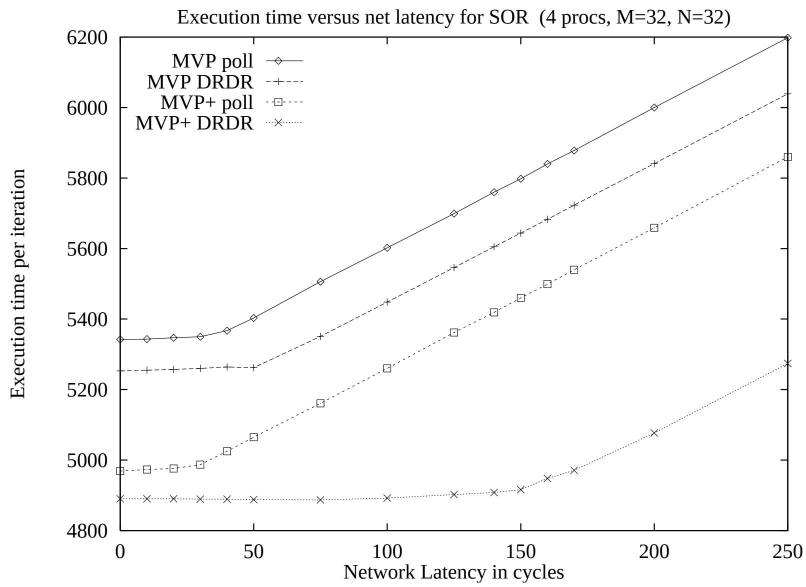

Figure 8: Execution Time versus Net Latency for `SOR` with 4 processors and $M = 32$, $N = 32$

4.5 Overall Simulation Conclusions

In general, DRDRs provide significant benefit, both in reducing the amount of communication overhead, and in increasing the latency tolerance.

Notice that the use of DRDRs does not always provide improved performance. In cases where the incoming message has been received before the first poll, the polling routine will return immediately, while the delayed-response device register version must set up some additional synthetic dependences.

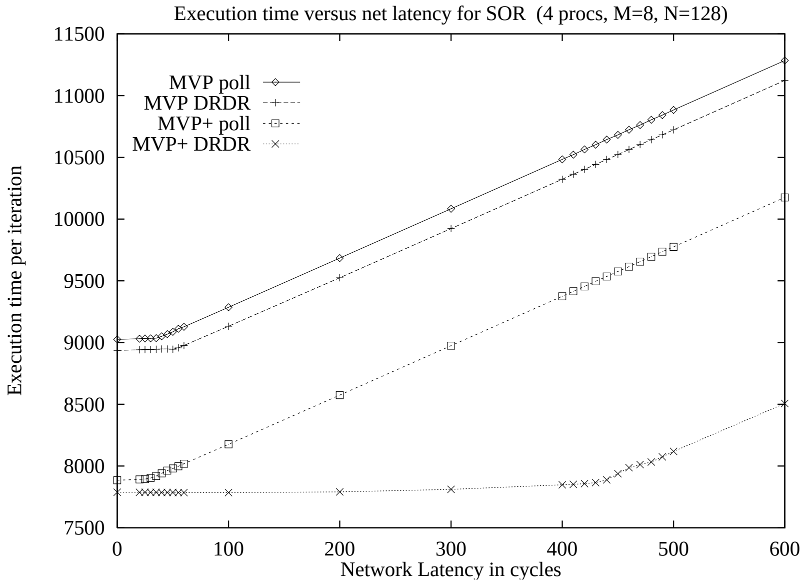

Figure 9: Execution Time versus Net Latency for `SOR` with 4 processors and $M = 8$, $N = 128$

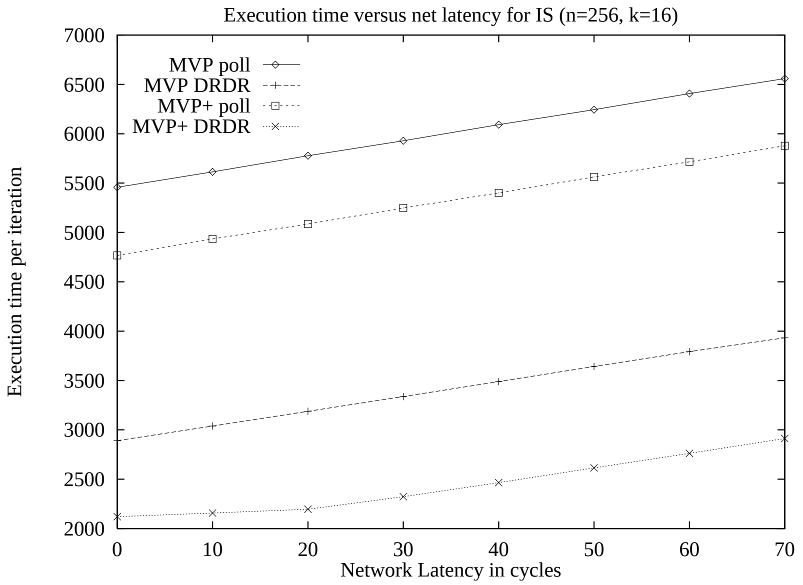

Figure 10: Execution Time versus Net Latency for `IS` with 16 processors and $N = 256$,$k = 16$

In addition, in our current model the use of synthetic dependences may end up restricting memory access parallelism by exacerbating the memory disambiguation problem. Notice, however, that since most traditional device-register interactions require `sync` instructions, the use of synthetic dependences does not prevent more memory access parallelism than a traditional implementation — any performance loss is due to the additional computational overhead of setting up the synthetic dependences, versus a single-instruction `sync`.

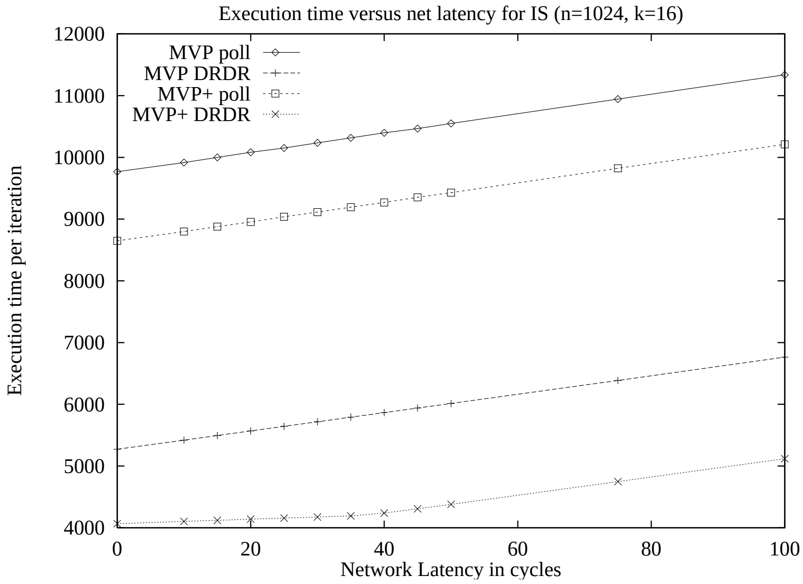

Figure 11: Execution Time versus Net Latency for IS with 16 processors and $N = 1024, k = 32$

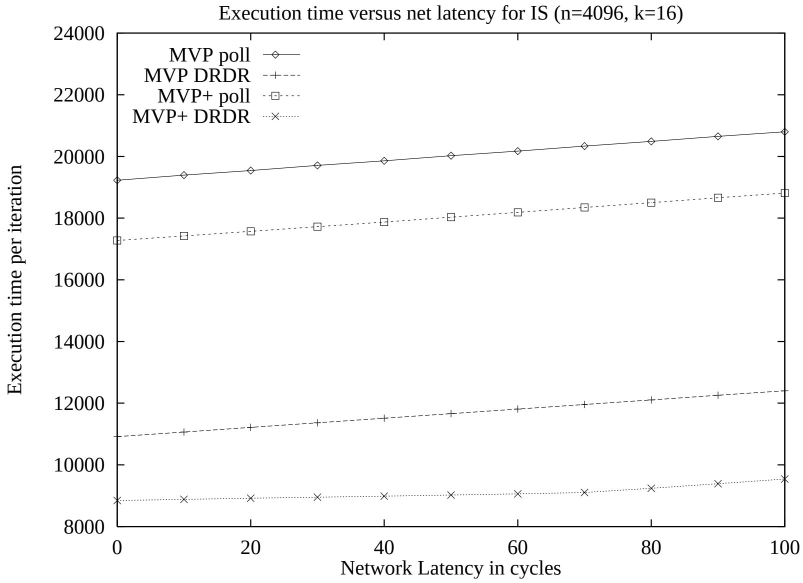

Figure 12: Execution Time versus Net Latency for IS with 16 processors and $N = 4096, k = 64$

Part of the overhead of synthetic dependences can be reduced by hand-tuning the library header files with inlined assembly code, to perform some obvious optimizations that gcc is unable to detect and shorten the critical path of the messaging routines by a few cycles. With messaging overheads around 20 to 30 cycles for a one-word receive-process-send transaction, saving a few cycles for the receive and the send will reduce the communication overhead significantly.

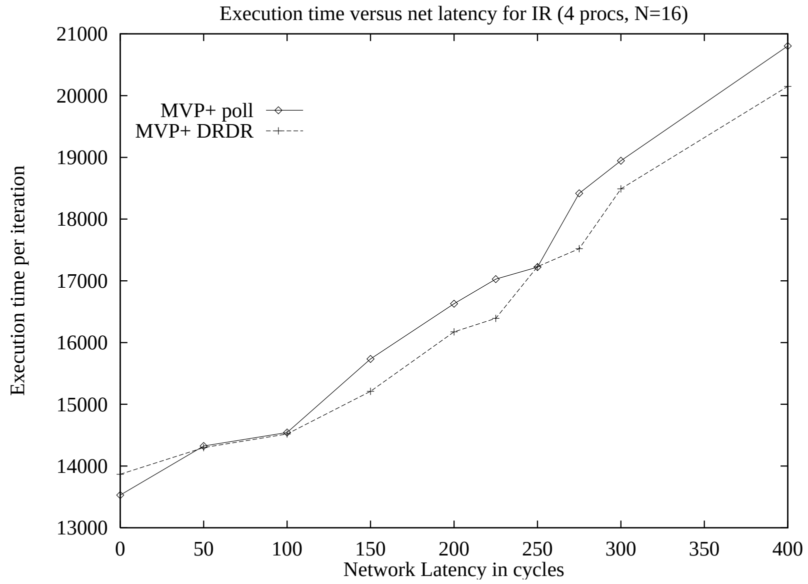

Figure 13: Execution Time versus Net Latency for `IR` with 4 processors and $N = 16$

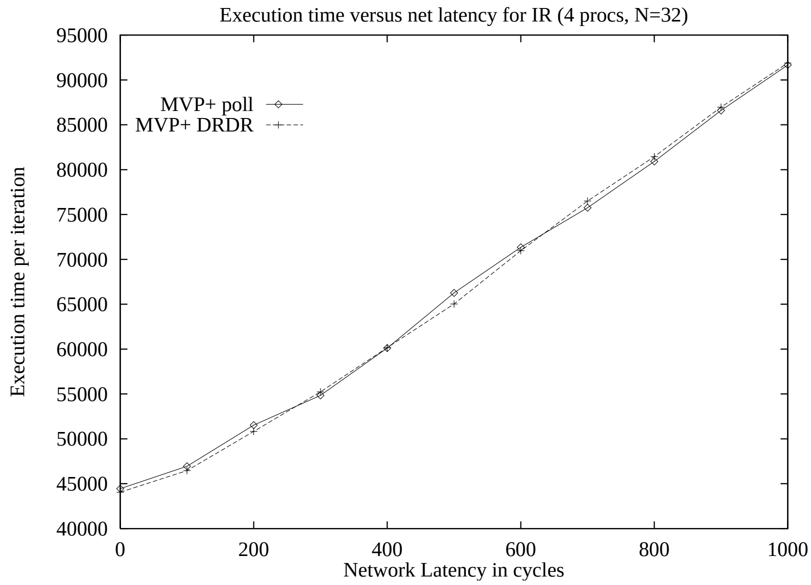

Figure 14: Execution Time versus Net Latency for `IR` with 4 processors and $N = 32$

5 Conclusions

Message passing requires synchronization between the processor and the network interface, to ensure that messages are not sent before they are composed, for example, or that data is not consumed before the appropriate message has arrived. In a superscalar processor with weak memory consistency, this synchronization has traditionally required memory barrier instructions (`syncs`) and spin-loop polling, both of which are substantial impediments to instruction level parallelism.

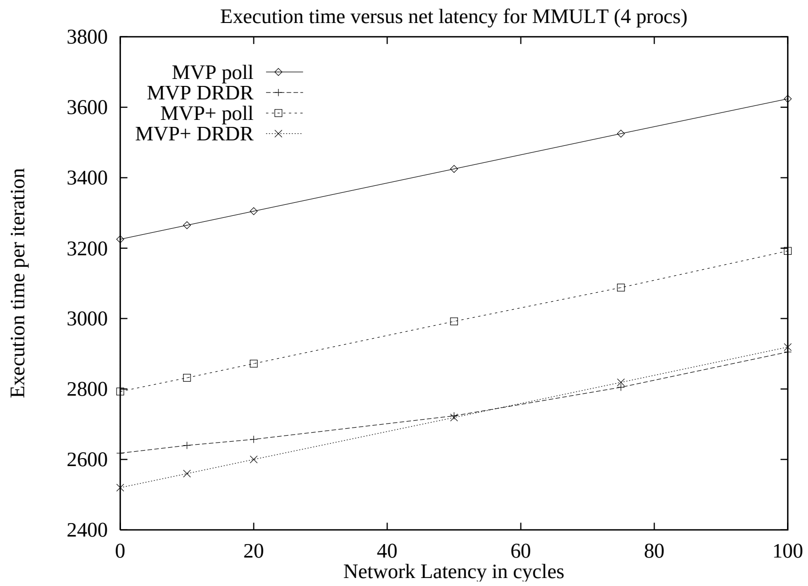

Figure 15: Execution Time versus Net Latency for `MMULT` with 4 processors

We have presented two techniques that allow an efficient dataflow interaction between the processor and the network interface without requiring a sophisticated coprocessor and without requiring any modifications to the processor's instruction set or microarchitecture. In addition, simulation results for a PVM-style message-passing library using these techniques were presented. The use of synthetic dependences and delayed-response device registers can provide significant improvement in reducing the communication overhead and improving latency tolerance, even when the high-level programming style is not changed.

References

[1] D. H. Bailey, E. Barszcz, and J. T. Barton *et al.*. "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, Vol 5, No. 3, 63–73, 1991.

[2] William Dally, J.A. Stuart Fiske, John Keen, Richard Lethin, Michael Noakes, Peter Nuth, Roy Davison, and Gregory Fyler. "The Message-Driven Processor," In *IEEE Micro*, April 1992.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[4] B. Grayson. Armadillo: A High-Performance Processor Simulator. Master's thesis, The University of Texas at Austin, 1996.

[5] Brian Grayson and Craig Chase. "The Challenges of Low-Overhead Message-Passing Communication Using Commodity Superscalar Processors," In *Proceedings of the ISCA Tenth International Conference on Parallel and Distributed Computing Systems*, pages 193–196, 1997.

[6] James Hicks, Mike Beckerle, Bob Greiner, Horace Thompson, Derek Chiou, Boon Ang, Andy Boughton, Tony Dahbura, and Venkat Natarajan. StarT Architecture White Paper. Technical Report MCRC-TR-45, Motorola Cambridge Research Center, 1995.

[7] Intel Corporation. *A Tour of the Pentium Pro Processor Microarchitecture.* Available on-line at `http://www.intel.com/procs/p6/p6white/p6white.htm`.

[8] John Kubiatowicz and Anant Agarwal. "Anatomy of a Message in the Alewife Multiprocessor," In *Proceedings of the Seventh International Conference on Supercomputing (ICS) 1993*, pages 195–206, 1993.

[9] Honghui Li, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. "Message Passing Versus Distributed Shared Memory on Networks of Workstations," In *Supercomputing '95*, 1995.

[10] Motorola Corporation. *PowerPC Microprocessor Family: The Programming Environments*, 1994.

[11] Shubhendu Mukherjee, Babak Falsafi, Mark Hill, and David Wood. "Coherent Network Interfaces for Fine-Grain Communication," In *Twenty-third International Symposium on Computer Architecture (ISCA 1996)*, pages 247–258, 1996.

[12] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. "The J-Machine Multicomputer: An Architectural Evaluation," In *Twentieth International Symposium on Computer Architecture (ISCA 1993)*, 1993.

[13] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 12–23, 1996.

[14] Gregory M. Papadopoulos and David E. Culler. "Monsoon: an Explicit Token-Store Architecture," In *Seventeenth International Symposium on Computer Architecture (ISCA 1990)*, 1990.

[15] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[16] Steven Scott. "Synchronization and Communication in the T3E Multiprocessor," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 26–36, 1996.

[17] Richard L. Sites. "Alpha AXP Architecture," *Digital Technical Journal*, Volume 4, No. 4, Special Issue 1992.

[18] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," In *Twentieth International Symposium on Computer Architecture (ISCA 1993)*, 1993.

[19] Robert van de Geijn and Jerrell Watts. "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, Vol. 9(4), 255–274, 1997.