The Challenges of Low-Overhead Message-Passing Communication Using Commodity Superscalar Processors

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

July 7, 1997

Abstract

Commodity microprocessors are generally used as the compute base for high performance parallel machines, including networks of workstations (NOWs) and parallel computers. In spite of recent advances, communication between processors remains relatively expensive. We present a message passing model and network interface design that reduces communication overhead nearly to the absolute limit. Unlike previous work, we assume a highly-parallel superscalar processing node with a weakly-consistent memory system. We show how traditional techniques (e.g. polling for message arrival) are poorly suited for superscalar processors, and introduce new synchronization methods that allow computation to proceed in a purely dataflow fashion, without modifications to the processor architecture. Messages are sent as soon as their data has been written, computation commences as soon as incoming data arrives, and local computation can be arbitrarily intermixed with communication subject only to the true data dependences that exist in the program.

1 Introduction

Virtually all contemporary parallel computers are based on high-performance commodity microprocessors. For example, the Cray T3E processor [10] uses Digital's Alpha 21164, the IBM SP series of machines is based on the RS/6000 processor family, the HP Exemplar uses HP's PA-RISC, etc. Recent demonstrations [1, 9] have shown that clusters of commodity workstations can deliver similar (or even superior) computation and communication performance to that of traditional parallel computers, which drives home the point that the "killer micro" is likely to continue to be at the heart of high performance computing for some time.

In this paper we extrapolate perhaps five or ten years along this trend and consider how the next generation of parallel machines should most effectively tap the incredible horsepower of the next generation of superscalar microprocessors. Two of the most important trends in microprocessor architecture are the increasing exploitation of instruction-level parallelism and the ever increasing CPU clock frequency. That is, processors are becoming dataflow engines, and memory (and the memory system) is becoming very slow. Future processors may be capable of executing literally hundreds of instructions in the same amount of time required to perform a reference to local memory!

*Brian Grayson is supported by a National Science Foundation Graduate Fellowship.

It is fairly well-known that using interrupts for synchronizing the arrival of data is unacceptably expensive on superscalar processors in terms of data locality and instruction-level parallelism. We show that even polling is unacceptably expensive because it introduces an unnecessary control dependence that inhibits instruction-level parallelism, and structures a loop around a high-latency operation (reading the device status). In contrast, we present an alternative form of synchronization that makes communication more "dataflow friendly." In our study, we assume a user-level, memory-mapped network interface, and rule out the possibility of modification to the micro-architecture or instruction set of the processor. Messages are sent and received by writing to pre-determined memory locations (device registers). Message status information is obtained similarly by reading. We allow for the possibility that device registers are cacheable [7] or even that the network interface is integrated into the cache controller. However, our techniques also have considerable benefit even when the latency to access the network interface is dozens of processor cycles.

Unlike earlier work, we assume that the processor provides a weak memory consistency model. In general, loads and stores may be performed in any order that is consistent with the true, anti and output data dependences of the single process running on the local node. Special "`sync`" instructions can be used in sections of code where a stronger memory ordering is desired. For clarity, we will assume a monolithic `sync` instruction that ensures all outstanding reads and writes complete before any subsequent reads or writes can be issued. Our work is equally applicable to machines with specialized synchronization instructions (*e.g.* acquires and releases [4]).

The major contributions of this paper are the following observations:

- Synchronization of the CPU with a network interface (or any other peripheral device) does not inherently impair instruction-level parallelism (ILP), although many implementations do.
- Efficient message handling does not require the use of either a) modifications to the instruction set or b) an intelligent "protocol processor" on the network interface.
- Communication overhead can be overlapped with computation if unnecessary control dependences and `sync` instructions are removed.
- Given a simple, cache-resident network interface, it is possible to receive, process, and respond to messages at a rate of one message every two cycles.

Our presentation is organized as follows. In Section 2 we provide background and discuss related work. Section 3 presents our model for a memory-mapped network interface, and describes our techniques for efficient synchronization. In Section 4 we demonstrate the severe effect of polling on instruction-level parallelism, and show that our technique permits virtually overhead-free communication. Section 5 concludes the paper.

2 Motivation

The focus of our research is high-performance fine-grain computation on parallel computers. Generally speaking, a computer capable of efficiently executing fine-grained programs is also capable of efficiently executing more coarsely grained programs; the exception occurs when there is insufficient bandwidth to sustain a coarse-grained computation. As we are interested primarily in the effective cost of synchronization on parallel computers, we assume infinite bandwidth is available in the network. Thus, fine-grained computation embodies all of our major challenges. We will also ignore the effects of network latency in this study – reducing network latency is orthogonal to interface design. Furthermore, our efforts to make synchronization more "dataflow friendly" naturally leads to a more latency tolerant architecture.

It should be noted that when we use the term "synchronization" we are referring to synchronization within a single node of a parallel computer, such as the synchronization required between data arriving at a node and that data being read by the processor.

2.1 Message-Passing Model and Related Work

Sending or receiving messages requires several phases: *production* of the message data for sends (*consumption* of the data for receives), *configuration* of the message, *initiation* or posting the message, and *notification* that a send or receive has completed. In Figure 1 we present a simplified typical set of message-passing routines, with these four phases of message-passing shown in comments. For simplicity, we have omitted arguments to the functions. Throughout this paper, these routines will be updated to reflect modifications to the communication model and the network interface.

```
send() {
  write data to sendBufferPtr          /*  Production  */
  configure message at sendConfigPtr   /*  Configuration  */
  sync
  initiate message                     /*  Initiation  */
}

send_wait() {
  while (NotDoneSending(pollReg)) ;    /*  Notification  */
}

recv_post() {
  configure message at recvConfigPtr   /*  Configuration  */
  sync
  initiate message                     /*  Initiation  */
}

recv() {
  while (NotDoneReceiving(pollReg)) ;  /*  Notification  */
  copy data from recvBufferPtr         /*  Consumption (Production)  */
}
```

Figure 1: “Traditional” message-passing code

Previous work in low-overhead message passing includes the following contributions. The MDP[2] processor provides very low-overhead message passing through network instructions and a tight coupling of the processor and the network interface. The CNI (Coherent Network Interface)[7] research examines techniques for leveraging the local coherence protocol support at each node to reduce memory traffic and software overhead. The Cray T3E[10] provides a large number (>500) of messaging registers and handles efficient remote Get and Put operations. Pakin *et al.*[9] discuss techniques for efficient messaging, including offloading some computation to the network coprocessor. Several researchers[5, 7] have discussed the impact of local coherence on data-passing operations.

However, several current high-performance microprocessors, such as the PowerPC and the DEC Alpha family, support only a weak consistency[3, 4] memory model. A weak consistency model allows memory references to be performed by the processor in a different ordering than that prescribed by the program. To date, no researchers have examined the impact of a weakly consistent architecture on the interactions between the processor and the network interface. To make matters worse, traditional message passing techniques directly conflict with some of the features that provide these processors with their high performance:

- The conditional branches inherent in polling impair instruction dispatch lookahead.
- The memory barriers or `sync` instructions, required when the traditional techniques are mapped to weaker consistency machines, restrict memory access reordering, which reduces the processor's ability to hide memory latency.

The remainder of this paper discusses new techniques that allow efficient, dataflow-style messaging on weakly consistent, superscalar architectures. These techniques remove polling loops and eliminate the need for memory barriers, and provide no unnecessary dependences (both data dependences or control dependences).

In this paper, we assume a node architecture comparable to what should be available within the next decade. In particular, the degree of parallelism will increase, split-transaction (non-blocking) caches and buses will be used throughout the memory hierarchy, and at least the upper levels of the memory hierarchy will be able to handle multiple independent memory accesses in a single cycle[1].

3 Communication Model

We assume a network interface (NI) that has several message "slots," each of which has a dedicated buffer of some fixed size and its own set of configuration and status registers. The configuration registers contain information to be placed into the message header (destination node, size of message), as well as values used by the NI and CPU. A partial example format of the configuration registers is presented in Figure 2. The exact format or number of configuration and status registers is of limited importance in this paper.

Byte Offset	Description
0	Destination node for sends, -1 for receives
4	Destination slot for sends, -1 for receives
8	Message length
12	Configuration write-count threshold (16-bit value)
14	Data write-count/read-count threshold (16-bit value)
16	CPU-Settable Flags (send/receive)
⋮	

Figure 2: An example of a set of configuration registers

In addition, we assume that the configuration registers are not implicitly cleared upon message completion. This allows repeated messages to the same node to reuse most of the configuration, rather than rewriting the entire configuration for each message.

3.1 Notification Synchronization

Notification synchronization is used, for example, to let a processor know that a message has been received, or that a message has been sent and its resources (buffer, slot) can be reused. The traditional high-performance solution to notification synchronization is polling[2]. A polling loop consists of an instruction (or instructions) to read the device status and a conditional branch, forming a loop. The processor continues to execute this loop until the value read from the device register indicates that the communication event has completed. However, polling in this fashion is fundamentally incompatible with a highly parallel superscalar processor. Consider the case where there are two potentially concurrent instructions I_1 and I_2, with I_1 occurring before the loop and I_2 occurring after the loop. Even if the processor supports both branch prediction and speculative execution, we can only obtain parallel execution of I_1 and I_2 if the processor predicts *exactly* the number of iterations that the loop will ultimately take. Each mispredicted branch results in the cancellation and ultimate re-execution of any speculatively executed instructions, such as I_2.

In practice, this restriction means that in order to overlap the execution of I_2 with I_1 we must ensure that the loop executes exactly once. That is, we must statically schedule the program code such that the first read of the device status register occurs after the event has already completed. In some cases, it may not be practical or even possible to schedule the synchronizing loop sufficiently "late," especially if the communication code spans one or more small subroutines.

[1]For example, the Alpha 21264 is capable of two level-1 cache accesses per cycle, achieved by running the cache twice as fast as the processor.

[2]In some cases, even the traditionally interrupt-oriented communication models like Active Messages and the message-driven communication on the J-Machine are implemented via polling for high performance [11]

Our messaging routines eliminate the need to statically schedule synchronization through the use of *delayed-response device registers* to replace polling loops. A delayed-response device register is a special status device register that contains information for one specific condition (as compared to traditional status registers, which contain a bitfield of several conditions). If a read request is received and the condition for this register is true, the delayed-response device register returns the "true" value. However, if the condition is currently false, the device does not respond until the condition becomes true, at which point it returns a "true" value. Notice that in a system with split-transaction buses, independent memory accesses can be performed during the time between the device register access and the reply. A modified `recv()` function is shown in Figure 3, where the polling spin-loop has been replaced by a single read of a delayed-response device register(a similar modification can be performed on `send_wait()`). This read will not complete until the message has been received.

```
recv() {
  recvstatus = RecvDoneStatusReg;
  copy data from RecvBufferPtr
}
```

Figure 3: Modified `recv()` code

Notice that, unlike traditional status registers, the delayed-response device register always returns the same value, a "true" value. Thus, the return value is not constrained to have any bits set or cleared. For convenience reasons explained below, most delayed-response device registers should return a value of 0.

3.1.1 Synthetic Dependences

The use of delayed-response device registers avoids an unnecessary conditional branch, increasing parallelism. However, the naive application of this technique removes all dependences between the arrival (or departure) of message data and computation using that data. In Figure 3, for example, the "copy data" instructions may be executed before the status access completes, that is, before the new message has been received.

Several techniques could be used to add the proper dependence, such as inserting a `sync` instruction before the data accesses. However, memory barriers or other traditional techniques provide a stronger dependence than needed, leading to inefficiency. If the delayed-response device register returns the value 0 (as suggested above)[3], we can simply add the return value to the pointer used to access the message-data buffer, as in the `recv()` code in Figure 4. Clearly, the processor can not dispatch the "copy data" code until the "new" value of `recvBufferPtr` is calculated, which depends on the read reply. We refer to this technique, where an explicit dependence has been added by modifying a pointer, as introducing a *synthetic dependence* into the program.

If we assume that messages are received inside a loop (as is often the case), we need to ensure that the message slot is not reconfigured too soon. Similarly, when sending messages, the reconfiguration and the data production should not be performed before the previous message for that slot has been successfully sent. The synthetic dependences for these actions have been introduced into the `recv()` and `send_wait()` code in Figure 4.

Combined, these two techniques (delayed-response device registers and synthetic dependence operations) allow the notification synchronization to be performed very efficiently: only a delayed-response device register load and an add instruction are needed (compared to a spin loop executed an unpredictable number of times). In addition, this notification code sequence does not impede future nondependent instruction decoding, dispatch, and completion, due to its lack of conditional branches.

3.2 Configuration-Initiation Synchronization

Configuration-initiation synchronization is used to prevent the initiating write from being performed (and thus the message being initiated) before the configuring writes have completed. Traditionally, a simple memory barrier has been used. This has the disadvantage of adding nonessential dependences: all outstanding memory accesses

[3]A similar application of adding zero to pointers to create dependences is discussed in [8].

```
recv() {
    /*  statusval will be 0.  */
  statusval = recvDoneStatusReg;
  recvBufferPtr += statusval;
  recvConfigPtr += statusval;
  copy data from recvBufferPtr
}

send_wait() {
    /*  statusval will be 0.  */
  statusval = sendDoneStatusReg;
  sendBufferPtr += statusval;
  sendConfigPtr += statusval;
}
```

Figure 4: Modified `recv()` and `send_wait()` code

must complete before the barrier is allowed to complete, and all future memory accesses are not allowed to be issued until the memory barrier completes. Other approaches are to allow or force uncacheable accesses to be sequentially ordered, or provide separate memory barriers for uncacheable accesses[6], both of which may still add unnecessary dependences, although typically not as many as a simple memory barrier. Ideally, we would like to perform this synchronization with no unessential dependences, and without sacrificing performance (in terms of latency, instruction overhead, and hardware complexity).

A novel technique that meets these goals in a weakly-consistent memory hierarchy is the *write-count* method. The write-count technique uses a counter in the NI to count the number of configuration writes that have been performed for each message "slot." For example, assume the configuration of a send message requires 8 words of information to be written, and message initiation is caused by writing one word to a special initiation device register. In this case, the network interface should initiate the message after the initiating write has been performed *and* after 8 words have been written to the configuration, whichever occurs later. If the initiating write arrives too early (*i.e.*, before 8 configuring writes), the NI sets a status bit to record that an initiation has been requested, and waits for a total of 8 configuring writes to arrive before actually performing the message initiation.

This technique avoids the unessential dependences of memory barriers or FIFO-ordered uncacheable accesses, adds no instruction or processor-resource overhead, and does not affect future nondependent instructions. The improved `recv_post()` code, where the `sync` has been removed, is presented in Figure 5. Unfortunately, the `sync` instruction in `send()` can not yet be removed – we still need to ensure that the data has all been written before the message is initiated. The next subsection discusses a way of finally removing this `sync`.

```
recv_post() {
  configure message at recvConfigPtr
  initiate message
}
```

Figure 5: Modified `recv_post()` code using the *write-count* technique to avoid the use of `sync`

3.3 Production-Initiation Synchronization

A similar synchronization problem arises when initiating send messages, as discussed above – the send should not be initiated until after all of the data writes have completed. Again, a memory barrier is traditionally used, with the addition of unessential dependences.

The write-count technique can also be used here, with a minor modification: unlike the message configuration, where the threshold write-count for message initiation was fixed, the message length is not fixed, so using a fixed initiation threshold of n writes is not sufficient. Clearly, the length of the message is part of the configuration, and is thus known to the NI as soon as the configuration has been written. The NI waits for that number of writes to its data buffer before it initiates the message. If the configuration has not yet been specified (and thus the length is unknown), the NI uses a threshold of "infinity" to prevent false message initiations; when the NI eventually processes the configuration write for the message length, it compares this length with the current write-counter value, and sends the message if they match. If not, it continues to wait for more writes.

Notice that the use of write-counting for both production-initiation synchronization and configuration-initiation synchronization allows the three components (data, configuration, initiation) of a send message to be written in a completely random order while still sending the message correctly. In addition, the initiation write is now redundant – if all of the data and a complete configuration are present in the NI (determined by looking at the write-counters), then the message can be initiated. A modified version of `send()` is presented in Figure 6.

```
send() {
  write data to sendBufferPtr
  configure message at sendConfigPtr
}
```

Figure 6: `send()` code that uses write-counters for implicit synchronization and implicit initiation

3.3.1 Consumption-Initiation Synchronization

For message receives, a new variation of the problem arises: in a weak-consistency system, how can we make sure that the initiation of a message is performed *after* all of the data from the previous message has been examined or copied? In traditional systems, the `sync` instruction between configuration and initiation enforces that any previous reads have been performed. However, we have replaced that `sync` with the write-count method for configuration-initiation synchronization, which provides no guarantee that earlier reads have all been performed.

We have chosen to use a *read-count* technique, similar to the write-count technique for sending messages, that prevents the initiation of a receive until the appropriate number of reads of the previous message have been performed. The read-count threshold for receiving messages is thus the "dual" of the write-count threshold for sending messages.

Note that the use of the read-count technique for consumption-initiation synchronization makes the initiating write unnecessary – a new receive operation will be enabled when the previous message's data has been referenced, and when a new configuration has been written. This initiating write has been removed in the code presented in Figure 7.

3.3.2 Speculative Memory Accesses

Notice that while our communication routines do not contain conditional branches, surrounding code undoubtedly will. Independent instructions from the predicted path of a conditional branch may be speculatively executed. However, we assume that speculative loads and stores to the NI's device registers are not allowed. Either the architecture does not speculatively execute any memory operations, or the region of memory for the NI device registers are marked appropriately in the memory management unit. Speculative loads, for example, would disrupt the use of read-counting. Speculative stores to *any* device registers should rarely, if at all, be allowed, due to the potentially devastating consequences. In addition, speculative memory accesses cause other problems, such as speculative TLB misses and speculative page faults.

3.4 Final Message-Passing Code

Taken together, the three techniques discussed here (delayed-response device registers, synthetic dependence operations, and write-counting/read-counting) allow one to implement message-passing on a weakly-consistent su-

```
recv_post() {
  configure message at recvConfigPtr
}

recv() {
  statusval = recvDoneStatusReg;
  recvBufferPtr += statusval;
  recvConfigPtr += statusval;
  copy data from recvBufferPtr
}

send() {
  write data to sendBufferPtr
  configure message at sendConfigPtr
}

send_wait () {
  statusval = sendDoneStatusReg;
  sendBufferPtr += statusval;
  sendConfigPtr += statusval;
}
```

Figure 7: Final version of the communication primitives

perscalar architecture without the use of explicit processor-controlled synchronization (polling, memory barriers). By eliminating nearly all "false" or nonessential latency and extraneous ILP-busting behavior, the entire system behaves more dataflow-like, and is thus more optimal. A final version of the four communication routines is listed in Figure 7. Note that the instruction counts of these procedures are short enough that many can, and should, be inlined by the compiler to further improve performance. No conditional branches or explicit `sync` instructions are performed, so the inlining of these allows much larger basic blocks around communication operations, leading to more compiler opportunity for optimization.

3.5 Simple Enhancements

We have examined several enhancements to the "base" model presented here. The following list contains enhancements, with a hint to one possible implementation in parentheses:

- Increasing effective bandwidth between the processor and NI by using wide reads and writes to the configuration or the data buffer, or using cacheable writes followed by an explicit cache flush[7] (*treat an 8-word wide write as 32 byte accesses during read or write counting*).
- Reads/writes of only part of the message buffer (*provide a write/read-count threshold configuration register that is distinct from the message-length register*)
- Partial updates of the configuration between messages (*allow the configuration write-threshold to be specified as part of the configuration*)
- Protection in a multiprogramming environment (*have each slot mapped to a different virtual page, and use standard virtual memory protection mechanisms to grant a single process access to only a subset of the physical slots*)

4 Analysis

A general, detailed algorithmic analysis of the performance improvement from these techniques is quite difficult, due to the importance of the characteristics of the code surrounding the communication operations. The degree to which we are successful in reducing effective communication overhead depends in part upon the number of independent instructions available to the CPU, and whether there are sufficient hardware resources available to overlap communication overhead with computation. Numerous parameters (memory bandwidth, degree of memory support for split transactions, latency of the memory hierarchy at each level, to name a few) play very important roles in the overall delivered performance. A simulation or prototype would only show the effect of a few sets of values for these parameters. For this paper, we consider an extremal point from the space of fine-grained computations, and we evaluate this code executing on a hypothetical processing node with infinite resources.

Assume we have an ∞-way superscalar processor with 1-cycle execution for most operations, with an unbounded number of integer units and load/store pipes and support for an unlimited number of simultaneously outstanding memory accesses. Also assume a cache capable of 1-cycle response, and an NI that is merged with the L1 cache (and thus NI device registers may be accessed in one cycle). Assume that the read/write-count threshold components of the configuration fit into a single word — the configuration-write-count threshold is in the upper 16 bits, and the production-write-count (or consumption-read-count) is in the lower 16 bits, as shown in Figure 2. Also, the NI "remembers" the configuration — to send a second message to the same node, only the threshold parts of the configuration need to be rewritten. Thus, a new message with the same configuration as the previous message can be reconfigured and initiated with a single 32-bit write (notice that this 32-bit write means the configuration write-count threshold should be set to 4 bytes).

It is well-known that much of the overhead of communication can be easily overlapped, or at least amortized, if messages are large or infrequent. In order to provide an unbiased and accurate comparison, then, we choose an application at the opposite extreme: a very small message size (one 32-bit word), and with very little computation (one add instruction). Our application is similar to a systolic computation: it repeatedly receives a word from one processor, adds it to a local variable, and forwards the result to another processor. For simplicity, all calculations will be integer-based, and we assume that all relevant variables can be held in registers. High-level code for this application is presented in Figure 8.

```
while (1) {
  receive val
  x = x + val;
  send x to next processor
}
```

Figure 8: High-level pseudo-code for a systolic-style calculation

The destination is constant, as is the sizes of the messages, so only the threshold values (that is, a single word of the configuration) need to be rewritten each iteration.

A traditional implementation is presented in Figure 9. Unfortunately, this code is highly serialized. On our theoretical architecture, the loop would require 8 cycles[4] between outgoing x update messages, as shown in the schedule in Figure 10. Notice that there can be very little overlap across iterations — the first polling load of the next iteration must occur after the last sync in the current iteration, so at most a single load is overlapped with the end of the previous iteration.

A similar analysis can be performed with a fixed cycle cost of d cycles to read or write an NI device register. This model corresponds to an architecture where the NI is placed lower in the memory hierarchy, perhaps even on the memory or I/O bus. The dataflow graph for this analysis is presented in Figure 11, where the number below the mnemonic identifies how many cycles that instruction requires. Notice that the longest cycle, and thus the time between successive iterations, is $4d + 4$. If $d = 1$, the iteration period is 8, as demonstrated in the schedule in Figure 10.

[4]If speculative loads are allowed, this can be reduced to 7 cycles, but requires the addition of one more `sync` instruction.

```
Loop:
 1:  load RecvDoneStatusReg → recvstatus   # Wait until data received
 2:  cmp recvstatus, DoneReceiving
 3:  bne Loop
 4:  load RecvBufferPtr → val              # Read val
 5:  sync                                  # Make sure read has completed
 6:  store receive-config → RecvConfigPtr  # Post the next receive
 7:  add x+val → x
SendPollLoop:
 8:  load SendDoneStatusReg → sendstatus   # Wait until previous send is done
 9:  cmp sendstatus, DoneSending
10:  bne SendPollLoop
11:  store x → SendBufferPtr               # Write x to message buffer
12:  sync                                  # Make sure the write has completed
13:  store send-config → SendConfigPtr     # Post the send
14:  jump to Loop                          # Repeat
```

Figure 9: Pseudo-assembly for a systolic computation

Instruction	Time t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9
1:Load	A								B	
2:Cmp		A								B
3:Branch			A							
4:Load				A						
5:Sync				A						
6:Store					A					
7:Add					A					
8:Load					A					
9:Cmp						A				
10:Branch							A			
11:Store								A		
12:Sync								A		
13:Store									A	
14:Jump								A		

Figure 10: Scheduling the pseudo-assembly of Figure 9. The best-case completion times for the first and second iteration are marked with A and B, respectively.

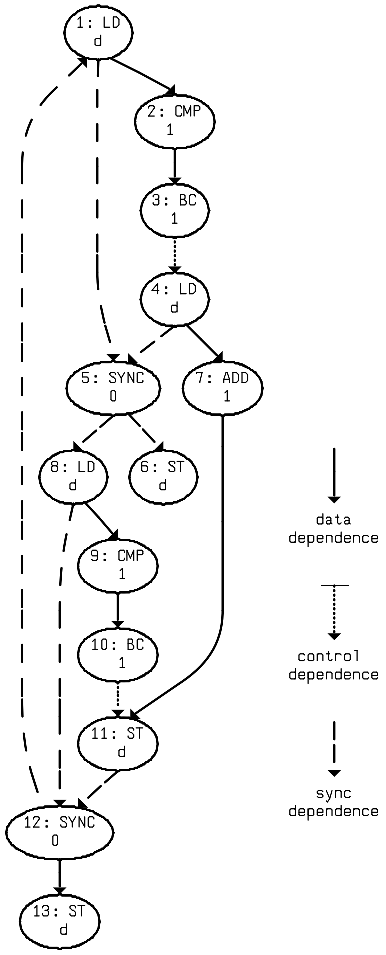

Figure 11: Dataflow graph for the pseudo-assembly of Figure 9. The cost of each instruction is listed under its mnemonic.

A version of the code that uses the techniques described in this paper is presented in Figure 12. One possible scheduling of this code on our ideal architecture is shown in Figure 13. With these techniques, the loop has been statically reduced from 14 instructions to 8 instructions, and from a throughput of one outgoing message every 8 cycles to one every 2 cycles. The performance improvement is due to both the removal of conditional branches (which allows more intra-iteration and cross-iteration overlap by enlarging the dispatch window) and the removal of memory-barrier instructions (which allows loads and stores to be performed out-of-order).

The dataflow graph is presented in Figure 14, where again the latency to perform a device register access has been changed to d instead of 1. The dataflow graph shows that the longest cycle is only $2d$, thus the initiation frequency is one iteration every $2d$ cycles. Even for NIs that are placed far away from the processor ($d >> 1$), then, these techniques reduce the iteration period by a factor of 2 over traditional techniques.

```
Loop:
1: load RecvStatusReg → recvstatus                 # Load recv and send status
2: load SendStatusReg → sendstatus
3: store receive-config → RecvConfigPtr+recvstatus  # Post next recv and send
4: store send-config → SendConfigPtr+sendstatus
5: load RecvBufferPtr+recvstatus → val             # Load val,
6: add x+val → x                                   # update x,
7: store x → SendBufferPtr+sendstatus              # and send x
8: jump to Loop
```

Figure 12: Pseudo-assembly for a systolic computation using several of the techniques of this paper

Instruction	Time t	t+1	t+2	t+3	t+4	t+5	t+6	t+7
1:load	A		B		C			
2:load			A		B		C	
3:store		A		B		C		
4:store			A			B		C
5:load		A		B		C		
6:add			A		B		C	
7:store				A		B		C
8:jump	A		B		C			

Figure 13: Scheduling the improved pseudo-assembly of Figure 12. The best-case completion times of the instructions in the first iteration are marked with A, the second iteration with B, and the third iteration with C.

Note that this code can still be improved in several ways. For example, if more slots are available for use, traditional double-buffer techniques would allow a best-case throughput of one outgoing message each cycle. In addition, a few simple modifications to the architecture would allow the posting writes to be eliminated, and another technique could remove the status checks, all without modifying the processor architecture.

5 Conclusion

Message passing requires synchronization between the processor and the network interface to ensure that a) data is not consumed before it arrives (or overwritten before it is sent), and b) that messages are not sent (received) before their configuration has stabilized. In a superscalar processor with weak memory consistency

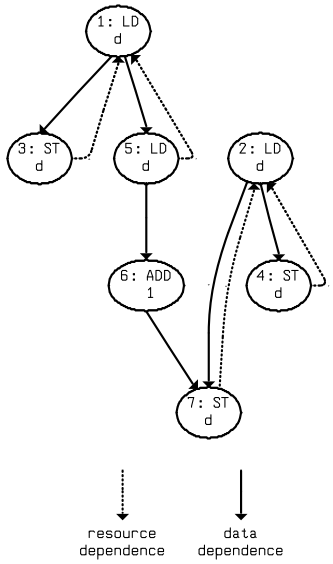

Figure 14: Dataflow graph for the pseudo-assembly of Figure 12

this synchronization has traditionally required memory barrier instructions (`syncs`) and conditional branches to implement polling. Both barriers and branches are substantial impediments to instruction level parallelism.

We have presented three techniques that allow efficient dataflow message-passing on weakly-consistent microprocessors, without requiring a sophisticated coprocessor and without requiring any modifications to the processor instruction set or microarchitecture. The first technique, delayed-response device registers, eliminates the conditional branch in a polling loop by delaying the response to any status read until after the message event (send/receive) has completed. We use the return value from our delayed response registers to introduce *synthetic dependences* into the program, thereby allowing instruction scheduling to be purely dataflow. Finally, we use *write-counting* to eliminate a race condition between writing data to the network interface and the initiation of a message event.

Compared to traditional polling, these techniques can achieve a factor of two to four reduction in communication overhead, depending upon the latency of accesses to the network interface. The message passing code is simpler, shorter and contains no conditional branches, allowing greater opportunity for both static and dynamic code scheduling and optimization. Although we discuss only message passing in this paper, similar techniques can be applied to synchronization (*e.g.* acquiring a lock before entering a critical section) to increase instruction-level parallelism in shared memory programs.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson et al. “A Case for NOW (Networks of Workstations),” *IEEE Micro*, feb 1995.

[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] Michel Dubois, Christoph Scheurich, and Fayé Briggs. “Memory Access Buffering in Multiprocessors,” In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[4] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessay. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[5] 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.

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

[7] Shubhendu Mukherjee, Babak Falsafi, Mark Hill, and David Wood. "Coherent Network Interfaces for Fine-Grain Communication," In *Proceedings of the 23th International Symposium on Computer Architecture*, pages 247–258, 1996.

[8] 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.

[9] Scott Pakin, Mario Lauria, and Andrew Chien. "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," In *Supercomputing '95*, 1995.

[10] 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.

[11] 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.