Responsiveness without Interrupts

Dejan Perkovic Peter J. Keleher
Department of Computer Science
University of Maryland
College Park, Maryland 20742
{dejanp|keleher}@cs.umd.edu

Recent advances in network and operating system technology allow applications to directly exploit gigabits of network communication per second. However, most such communication interfaces provide no efficient means of asynchronous notification to processes of incoming messages. Instead, applications are expected to poll frequently. In addition to the burden placed on the programmer, polls are rarely cheap enough to sprinkle indiscriminately in all program loops. We show that the resulting delay in handling messages can be very detrimental to application performance.

We characterize the communication behavior of our applications and show that the majority of notification delay is caused by a relatively small number of large delays. This result explains why coarse-grained timeout schemes are able to produce acceptable performance on network interfaces that lack explicit interrupt notifications.

We then study two methods of addressing this notification delay. First, we show that multi-threading schemes intended to hide other forms of communication delay are also fairly effective at hiding notification delay. Second, we evaluate several strategies for minimizing notification delay directly though judicious insertion of network polls. Both techniques produce performance comparable to fast software interrupts, especially important on communication systems that do not provide interrupt capabilities, such as PVM or MPI.

1. INTRODUCTION

Recent networks have shown enormous gains in bandwidth. Systems sold today range from Fast Ethernet at 100 Mbits/second, to Gigabit Ethernet and the Myrinet [4] at 1 Gigabit/second. Zero-copy protocols and new cluster specifications like VIA [1] often allow this bandwidth to be accessible between user-level processes. While message latency has not totally kept pace with bandwidth, the use of memory-mapped interface [3, 12] has allowed hardware latencies to also be reduced dramatically. For example, the BIP software developed at INRIA [22] allows round-trip latencies of small messages between user-level applications on the order of 10 *m*secs over Myrinet hardware.

Unfortunately, this improved performance is not uniformly available to different types of applications. High-performance communication infrastructures built on top of such networks, such as PVM and MPI, provide good performance for applications in which processes communicate at the same time. However, they provide no way to asynchronously notify applications that messages have arrived. Instead, they rely on applications polling frequently enough to ensure timely message handling. By default, polls of communication interfaces usually occur only when new messages are sent. This approach is even used by most implementations of Active Messages [8], whose interface suggests exactly the opposite.

While polling approaches are sufficient for applications that communicate only in parallel, we argue that they perform poorly for applications that have asynchronous request-response communication patterns, including most client-server applications. The reason is that asynchronous requests may languish at the network interface or in protocol buffers for long periods of time before being noticed.

We evaluate the use of two well-known techniques in addressing this *notification delay*. The first technique is multi-threading. Multi-threading has been used in many systems to mask communication costs [17, 20, 28]. The difference here is that notification delay may be many times larger than the base communication costs, and therefore multi-threading may be less effective. The second technique is to use a binary code re-writer to insert poll commands into the application code. Again, this technique has been used before in other contexts [9, 24].

We are not proposing either of these techniques as a more desirable alternative to providing message-notification mechanisms directly in the communication substrate. Indeed, our results show that even relatively slow and expensive interrupts provide adequate performance. We are only interested in providing acceptable communication behavior in situations where such notifications are not available.

1.1 Contributions

The first contribution of this paper is a characterization of the delays actually observed in a suite of applications. We show that the majority of notification delays result from a small number of large delays. These delays can dominate any gains achieved through use of new network technologies. The impact of these delays can be considerable. Our applications averaged more than 31% slower without interrupts than with them. This result argues that the problem is serious, and needs to be addressed either by including interrupts in emerging standards, or through use of the techniques discussed below.

The second contribution of this paper is an investigation of two popular approaches to improving the performance of applications in the presence of communication delays. The first is to tolerate remote request latency on the requester side, rather than trying to ensure that the latency does not incur in the first place. We tolerate latency by using multi-threading to perform other work when one thread blocks on a remote request. The use of multi-threading to hide remote communication has been studied before [17, 20, 28]. Our emphasis is different in that we are looking at longer delays relative to run lengths that we can use to hide the latencies. One advantage of multi-threading is that its effectiveness is not necessarily affected by loop and program structure. However, such parallelism is not always available, and the cost of switching between even user-level threads is often an order of magnitude more costly than polling a network interface. Nonetheless, one contribution of this paper is our finding that multi-threading is an effective substitute for interrupts for our application domain.

The second approach is to increase the frequency of polling in an attempt to ensure that messages are handled in a timely fashion. The standard polling procedure [6, 8] is to poll after each message send. The advantage is that these polls can be done inside the communication library, transparently to the application. This works well if messages are sent frequently (in the absence of contention) or in lockstep. However, incoming messages can wait too long if outgoing messages are sent only infrequently by local processes.

We used the ATOM [26] binary re-writer to insert calls to polling routines at several different levels in the program structure, ranging from once per function to polling in every basic-block. Good placement of polling calls is a non-trivial problem because its effectiveness can be very dependent on loop behavior. For example, some applications spend the majority of their time in heavily nested loops, while others spend the majority of their time in a single, small loop. While the cost of polling is small, it usually requires an I/O bus transaction. Hence, inserting polling calls into innermost loops might add unacceptable overhead for an application whose performance is dominated by a small (but frequently executed) loop. On the other hand, not inserting polls into heavily executed loops might result in too long of a delay. Our results show that the best approach is to use the processor cycle counter to limit the frequency of polls to a preset frequency.

We studied the performance of five standard shared-memory applications using SimCVM, an instruction level simulator of the CVM distributed shared memory system [14] running on a cluster of commodity workstations.

The rest of the paper is as follows. Section 2 describes the application domain that we are studying in detail. Section 3 describes our simulation environment. Section 4 characterizes the communication behavior of our applications. Section 5 discusses the performance of the multi-threading approach and Section 6 discusses the performance of the polling approach. Finally, Section 7 discusses related work and Section 8 concludes.

2. TARGET APPLICATION DOMAIN

We are interested in applications whose performance is tightly coupled to the *notification delay* incurred by requests sent between the application's processes. The notification delay is the delay between the time that a request becomes available at a node, and when the node actually starts to service the request.

For example, consider a database server. Each client's performance is clearly dependent on the notification delay of its requests at the server. However, the aggregate performance of the system is not, as message servicing is only delayed in order to service other requests. The order in which requests are serviced might not matter from the perspective of the server's overall performance.

By contrast, consider a distributed shared memory (DSM) system [2, 13-15]. DSMs support the abstraction of shared memory to parallel applications running on top of commodity workstations and networks. DSM application performance *is* sensitive to notification delay. Consider a single process. Before the next global barrier, the process must perform some local computation and

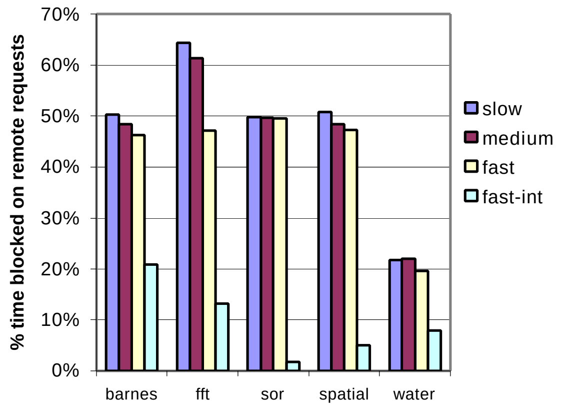


Figure 1: % time blocked on remote requests

service any incoming data requests. It does not matter to the local process in which order these actions are accomplished. However, each data requester is blocked from performing useful local work until its request has been serviced. Hence, the best overall system performance is achieved by servicing requests as soon as possible, even at the expense of postponing local computation.

This sensitivity to notification delay is not specific to DSM systems. Notification delay becomes important whenever a group of processes communicate to perform a common job, such that the processes synchronously stall on requests to each other at non-predetermined times. Examples relevant to industry include systems built with CORBA or any of the many distributed object systems built with C++.

Notification delay has become increasingly important as other sources of network overhead have been reduced or eliminated. For example, recent commodity networks such as Myrinet [4] and the Memory Channel [12] provide user-space to user-space latencies of under ten microseconds, with bandwidths near a Gb/sec. However, such systems rarely provide a means other than polling for notifying processes of incoming messages. This lack can severely impact a parallel applications performance.

As a quick example, Figure 1 shows time blocked on remote requests as a percentage of total execution time. These numbers reflect eight-processor runs on three different networks, with polling only after message sends. Blocking time with fast interrupts on the "medium" network are shown for comparison. The applications and configuration specifics are discussed below. However, blocking time is clearly much larger with any of the polling cases than for the interrupt case. Furthermore, the relatively small improvement between the 'slow' network and the 'fast' network implies that the majority of this time consists of the notification delay, rather than communication overhead.

Notification delay can also degrade performance indirectly by delaying synchronization releases. This is only applicable to applications that use peer-to-peer synchronization, such as exclusive locks. Some of these applications perform well with the base polling model because they communicate frequently.

3. SIMULATION ENVIRONMENT

The simulator used in this study is SimCVM, an execution-based simulator that models the CVM (Coherent Virtual Machine) [7] software DSM. CVM is a user-level library that features a set of base classes that provide a framework for implementing specific DSM protocols. These classes include a generic protocol class, a

Source	Overhead (cycles)
Mprotect	400
Block interrupts	400
Thread switch	400
Poll	20
Check for poll	5
Fast interrupt cost	1000
Bcopy	4.5 cycles/word
Diff creation	600 + 3*#words/page + .87/diff-bytes
Twin creation	33 + 3*#words/page
Diff application	25 + 1.2 * #bytes
Slow network	8000 cycles per send/rcv, 155 Mb/s,
Medium network	8000 cycles per send/rcv, 600 Mb/s
Fast Network	400 cycles per send/rcv, 1 Gb/s
Processor speed	200 MHz

Table 1: Simulation parameters

class that allows a protocol to hook into the virtual memory system to set page permissions and handle page faults, and efficient, reliable message-passing facilities based on UDP. New protocols are added by deriving classes from these base classes. The fact that all protocols implemented under CVM use the same underlying support for functions such as handling virtual memory and message passing allows them to be fairly compared.

The simulator consists of a modified version of the CVM library and a set of instrumentation code that is added to an application using the ATOM [8] binary-rewriting tool. The simulator directly executes the code for both the applications and for CVM's consistency protocol. The simulator runs as a single UNIX process, context-switching between multiple user-level threads to simulate multiple processes. The instrumentation code maintains a processor cycle count for each virtual processor, and handles switching between threads. The cost of each basic block is calculated statically and passed to an instrumentation routine inserted into the end of each basic block. The function of this routine is to update the logical time of the current node. The simulator models processor detail only down to cycle counts based on instruction type; we do not simulate pipelines or multiple issues per cycle. Therefore, it is not fair to say that our simulator reflects any single processor. However, we have roughly based our processor cycle counts and instruction costs on Linux running on a 200 MHz Pentium processor.

Operations such as signal handlers (used to trap write faults and to inform the system of incoming requests) and communication primitives are assigned costs, expressed in cycle times, that can be varied by parameters passed to the simulator when an application is run. A message leaving a virtual processor is tagged with an arrival time based on the cycle count of the sender and the assigned message costs, and the message passing facility in the modified CVM library delivers the message to the destination virtual processor when it reaches the message's arrival time.

The simulator also uses instrumentation to catch shared reads and writes, allowing it to simulate page faults and to record which words have been changed by which processors. This information allows us to know when a processor has not yet propagated a change to another processor through the DSM protocol, even though in our simulator the "processors" are threads that actually share the same physical copy of all shared data.

SimCVM instruments most loads and stores to check whether they reference shared memory. Instrumented accesses that violate current page protections are vectored to CVM's page fault entry point.

Apps	Input	Mes-	Kbytes	Loop	Function
Barnes	16k bodies	6154	5397	51	238
FFT	64x64x64	2712	1247	76	386
SOR	2k x 1k	543	695	105	246
Spatial	1000	2452	3988	365	874
Water	512	5520	3398	236	1029

Table 2: Application characteristics

We simulate all interactions with the operating system and all communication at a high level. Contention is modeled at message destinations, but not on the interconnect itself. We use a queue to represent each communication channel. Incoming messages are timestamped according to the logical time at which they should be received. The first poll after this time sees the message. Additional messages that arrive during the execution of a message handler are handled when the first completes. When modeling interrupt-based approaches, the incoming message queue is checked after each basic block. We simulated three different network configurations, chosen to reflect current UDP over ATM connections ('slow'), an OC-12 network ('medium'), and gigabit Myrinet [4] running the very-low-latency Bip [22] software package ('fast').

Table 1 shows the constants used throughout the paper. These constants were chosen to reflect the performance of CVM running on Linux 2.0.32 on 200 MHz Pentium Gateway machines.

The applications used in this study are Barnes, FFT, Water, and Spatial from SPLASH-2 [29], and a red-black SOR. Table 2 summarizes inputs and characteristics. "Messages/sec" is the total number of messages sent per second with the fast interrupt configuration discussed below. Loop and function lengths are static cycle counts, assuming only a single iteration per loop.

Cache and TLB effects

Our simulator can run with or without a simulation of the first-level cache. We model a 32 KByte direct-mapped level-1 cache, a 32-entry TLB, and assume a "perfect" second-level cache (i.e. the standard assumption of no misses in the second level cache). Since we simulate a software DSM, neither cache has coherence misses.

Table 3 shows application speedups with a fixed cost of 2 cycles per memory access, and with a variable cost where 6 cycles are charged for each cache miss and 30 cycles for each TLB miss [23]. We evaluated speedups for one and four threads for the slow and fast networks, with and without interrupts. The polling scheme polls only at messages sends.

Overall speedups for the cache runs are just slightly higher than for the fixed-cost runs discussed below. Cache speedups are generally higher for polling than with fixed costs, while the reverse is true for interrupts. This can be explained by noting that interrupts often arrive before a thread would normally have blocked, resulting in shortened run lengths. Shortened run lengths result in lower hit rates, and therefore lower performance.

One motivation behind the cache comparison is to determine the extent to which multi-threading hurts cache performance. This would be implied if the fixed-cost numbers improved more from one to four threads than did the cache numbers. The bottom line is that we do not see such a trend, implying that multi-threading can be used without concern for negative cache effects.

app (cache)		polling				interrupts			
		slow		fast		slow		fast	
		1	4	1	4	1	4	1	4
barnes	n	3.31	4.10	3.95	5.18	4.71	4.82	6.02	6.05
	y	3.38	4.15	3.91	5.29	4.59	4.70	5.94	5.99
fft	n	3.04	4.46	4.24	5.32	5.04	5.09	6.65	6.49
	y	2.88	4.10	3.43	5.72	4.96	4.92	6.58	6.47
sor	n	3.94	7.07	3.99	6.80	7.63	7.57	7.83	7.75
	y	3.92	7.15	3.98	6.74	7.53	7.43	7.77	7.67
spatial	n	3.62	5.74	4.04	5.85	6.64	6.36	7.38	6.61
	y	3.79	5.82	4.05	6.00	6.54	6.29	7.35	6.60
water	n	5.60	5.65	6.19	6.26	6.36	6.57	7.19	7.18
	y	5.64	5.46	5.72	6.11	6.31	6.56	7.14	7.20

Table 3: Speedups w/ and w/o cache simulation

The data shown here does not reflect operating system traps or interrupts, which would uniformly degrade cache performance. The effect of the operating system on cache performance is often approximated by assuming a smaller cache size. However, we repeated the above measurements for both 16 Kbyte caches and 256 Kbyte caches and observed the same general trends.

Given the similarity of the results, we turned the (slow) cache simulation off for the remainder of our experiments.

4. COMMUNICATION PATTERNS

Figure 1 shows that polling-based applications spend a large amount of time blocked. We know that large blocking times directly reduce potential speedups, but how sensitive are application speedups to the blocking time of individual messages? We modified SimCVM so that we could artificially vary notification delays seen by incoming message. Each arriving message is delayed in the incoming message queue until a specified interval has elapsed, and then delivered as soon as possible. Message delivery is further delayed only if the eventual message arrival occurs during execution of protocol or operating system code. The duration of these events is relatively short compared to the delays that we are investigating. Therefore, the majority of message delays center around our desired values.

Figure 2 shows application speedups as this delay interval is varied from 0 to 2000 *m*secs. Note that all messages have approximately the same notification delay for each data point. By contrast, during real executions the notification delay depends on the interleaving of computation and communication. All runs use the 'medium' network configurations, and single-threaded applications. Speedups do not drop off sharply, even at relatively high notification delays of 2000 microseconds. These results imply

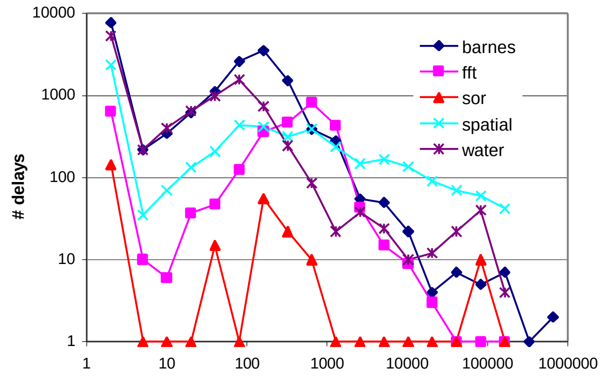


Figure 3: Observed notification delay (*m*secs)

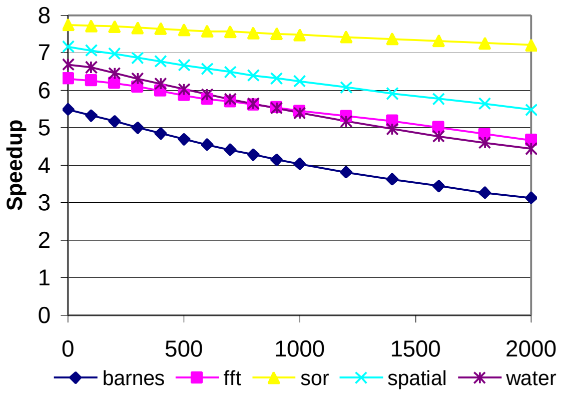


Figure 2: Notification delay (*m*secs)

that polling must be imposing long delays in order to produce the large blocking times shown in Figure 1.

Figure 3 shows a log-log plot of observed notification delays for the base, single-threaded scheme. Recall that the base scheme does not use interrupts, and polls only at message sends. The majority of delays are short, but the tail of the distribution is quite long. Figure 4 presents the same data in a plot of cumulative total delay. This data shows that the majority of notification delay is caused by a relatively small number of large delays. As an example, only about 14% of total delay for water is caused by delays of less than 10 milliseconds. In an even more extreme example, less than 3% of delay for SOR is caused by individual delays of less than 30 milliseconds. These results suggest that cutting off the expensive "tail" of the delay distribution can eliminate the majority of the delay. This is the approach taken by hardware schemes the Polling Watchdog [18].

We plotted the distribution of message sends (and hence polls) in order to see if they explain the large notification delays. Figure 6 shows the duration between consecutive message sends for each application. For the base polling case, this is also the extent of application execution between consecutive polls. Note that unlike Figure 4, these results are not cumulative. For example, Figure 6 shows that, about 25% of the time, the interval between consecutive sends for sor is approximately 100 *m*secs.

Why do these long notifications exist? In SOR, the reason is that each iteration is constructed in such a way that the barrier master sends a data request only at the end of each iteration, while all other nodes send requests at both the iteration's beginning and end. Requests arriving at the beginning of an iteration will only be seen by the barrier master at the completion of all of its work. Hence, the computation performed by the master is serialized

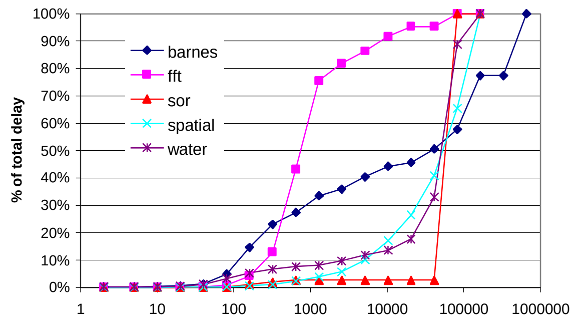


Figure 4: Percentage of cumulative delay (*m*secs)

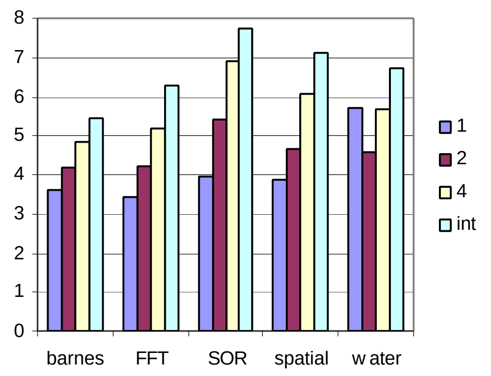


Figure 5: Multi-threading speedup

with respect to the slaves. While this is an extreme example, it provides a graphic illustration of the general problems inherent to achieving acceptable speedups in polling environments.

5. MULTI-THREADING

Multi-threading is a technique for *tolerating* remote message latencies rather than minimizing them. As with hardware systems, multi-threading in software systems can be used to reduce the costs of remote requests by switching threads when the current thread blocks. Assuming that the second thread has work to perform, some or all of the latency of the first thread's request can be overlapped with the computation performed by the second.

In most cases, multi-threading can be added transparently to our DSM applications. The applications can already be parameterized at the command line to use different numbers of processors. Data seen by the applications consists entirely of the stack and the shared segment (the "lightweight thread model" assumed by SPLASH-2). Hence, whether threads are on the same or distinct nodes is invisible to the application.

Figure 5 shows eight-processor speedups from runs with one, two, and four threads per node, plus a single-threaded run using fast interrupts. Message notification for the threading runs is accomplished through our base polling model: the network is polled after each message send. Our polling cost is set at 20 cycles, just slightly more than the cost of a PCI bus transaction to read a device register. The dip in performance for water at two threads is caused by a loss of locality when work is broken into smaller units [28].

The runs in

Figure 5 reflect the 'medium' network model. Performance differentials with the other two network models are qualitatively similar, with slightly better multi-threading performance for 'fast', and slightly worse for 'slow'. Thread switch costs are modeled as the 2 *m*secs incurred by the NewThreads [11] thread package on our system. Fast interrupts are included to provide a point of comparison. Our fast interrupts deliver signals and return in 5 *m*secs, considerably faster than the 100 *m*secs or so required by most current systems. This cost reflects a highly optimized implementation, such as that described by Thekkath [27].

Overall, multi-threading improves polling speedup from an average of 62% of the fast interrupt speedup, to 86%. As multi-threading can be used to hide all types of delay, not just notification delay, the multi-threading numbers could be even higher. Previous studies have shown that multi-threading can improve the performance of parallel DSM applications by nearly 50% [20, 28], speedups not reflected in our numbers. However, the former study used source modification to get around reduction-like operations, and much of the latter's speedup was due to cold misses. Our study focuses on steady-state behavior, and we did not modify application source.

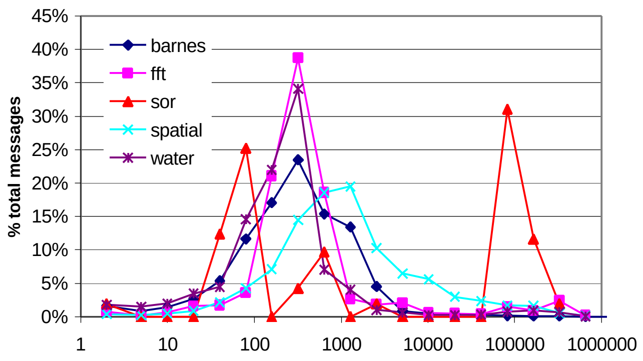


Figure 6: Gaps in sending messages (*m*secs)

6. POLLING

We are investigating polling and multi-threading only because many communication systems fail to provide interrupt mechanisms. However, polling does have advantages. The major advantage of polling is its cost: a poll is typically one to two orders of magnitude cheaper than an interrupt. Secondarily, the use of polling eliminates the need for interrupt mask manipulation because messages can be prevented from arriving merely by not issuing polls. This simplifies the construction of critical sections because messages can only arrive at specific locations in the code.

Another difference between polling and interrupts is that interrupt processing occupies the processor. Delays in issuing polls do not imply wasted cycles because the processor is presumably doing something else during the delay interval. Even if the notification delay were similar in both the polling and interrupt cases, the polling approach would be able to free more cycles for local computation.

The question of where to insert polls reflects a tension between the need to reduce runtime overhead (fewer polls), and the need to reduce notification delay (more polls). By default, most systems that use polling either poll after a message is sent [6, 8] or rely on the application programmer to insert polls at appropriate places (MPI and PVM). Neither is a good general solution. The former relies on the frequency and distribution of incoming messages to match that of outgoing messages. The latter puts the burden on the programmer and is error-prone. Such errors do not affect correctness, but can result in long notification delays or excessive overhead.

We assume polls can be inserted into the compiled code through the use of ATOM [26], or similar tools. ATOM is a binary code re-writer that can be used to walk the call graph, and to identify basic blocks, functions, and loop boundaries. All are potential sites for polls, but have distinct drawbacks. The default version of ATOM will only insert procedure calls to instrumentation routines located elsewhere in the code. However, XATOM, the experimental version of ATOM used in Shasta [24], can insert arbitrary code sequences. ATOM (and XATOM) currently run only on DEC Alphas running Digital Unix, but a port is underway to the i386 platform. Additionally, the Etch project [16] at Washington has similar functionality.

One disadvantage of inserting additional polls into application code is that application programmers can no longer assume that code between message sends is executed atomically. However, our technique could be extended to respect a software flag that signals when code should be executed atomically. Use of this flag should not change our performance advantages.

We model polls using the cost of an I/O transaction. We use an aggressive value of 20 cycles. This value likely understates the cost significantly on current high-end PC hardware. While cheap compared to a software interrupt, this is expensive compared to a typical basic block or loop iteration. The "loop length" and "function length" columns of Table 2 show the length in cycles of average loop iterations and average functions. For both cases, we assume that any nested loops execute only a single iteration. Inserting a poll into each loop iteration would add overhead from 5% to 39% for these applications. On the other hand, not instrumenting even small loops might result in long notification delays if the loop iterates a large number of times. All available messages are handled at each poll.

"Check" routines

We amortize the cost of polling by using a cheap "check sequence" to poll only when the interval since the last poll exceeds a preset threshold. The check sequence relies on using on an on-chip cycle counter that can be addressed as a register and accessed in one or two cycles, such as on the Pentium, Alpha, or Cray processors.

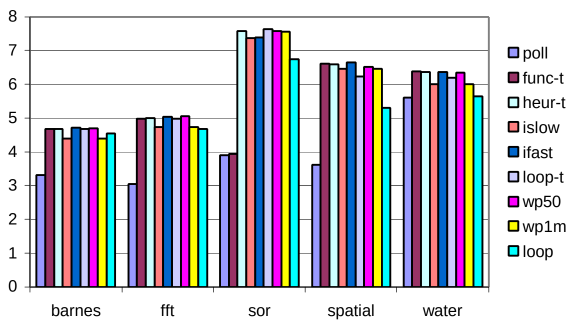
Our model check sequence is three instructions long, consisting of a read of the cycle counter, a comparison to a saved polling threshold, and a taken branch over the polling code. Given the combination of sophisticated branch prediction and branch target prediction techniques on most current processors, the branch is unlikely to cost much more than a cycle on average. The code sequence could take even fewer cycles with branch folding. We are currently assigning five cycles as a conservative estimate of the cost of the check routines.

The check sequence does require a dedicated register to hold the polling threshold, and a temporary register to hold the results of the comparison. Code generators can often be directed to ignore specific registers.

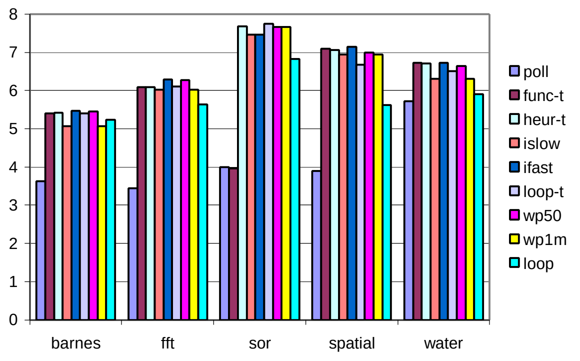
For our applications, a five-cycle check sequence adds approximately 10% loop overhead in the worst case, and less than 1% in the best case. Set against this is significantly decreased notification delay.

Polling Performance

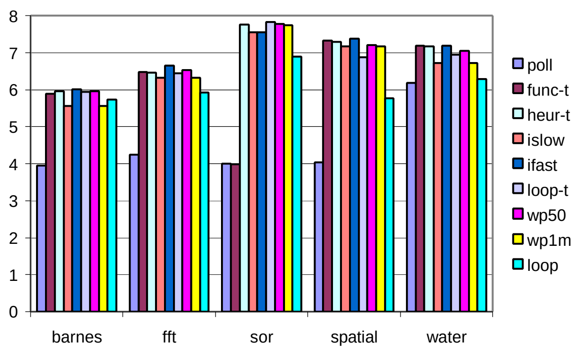
Figure 7 shows processor speedups for slow, medium, and fast simulated networks. Our first simulated configuration is 'base', which polls only at message sends. 'Func-t' places checks only at the beginning of functions, with a polling threshold of 5000 cycles. 'Heur-t' inserts checks only in functions and the outer loops of loop nests, and again uses a threshold of 5000 cycles. This may result in large notification delays if applications spend most of their time in singly-nested loops. However, we have not often observed this in practice. A "loop nest" must be entirely inside a



(slow)



(medium)



(fast)

Figure 7: Speedups with different networks

single function to be identified as such by our heuristic. 'Loop-t' inserts checks into all loops, while 'loop' inserts polls into all loops. 'Ifast' and 'islow' are our simulations of fast (5 *m*sec) and slow (100 µsec) interrupts. Note that we investigated several different thresholds for 'func-t', 'heur-t', and 'loop-t', and found comparable performance.

Finally, 'wp50' and 'wp1m' simulate the "polling watchdog" [18], with timeouts of 50 microseconds and 1 millisecond, respectively. The Polling Watchdog combines interrupts and polling by causing a timer interrupt to trigger if no polls happen in a specified time. The benefit is that polling can be inserted conservatively (i.e. not in loops). Interrupts occur only if the application spends excessive time in a loop. Our simulation of the watchdog generates a 'fast' software interrupt any time a message is not detected within the appropriate interval of time after arriving at a node. The polling watchdog requires either additional hardware or a programmable network card, such as FORE's ATM cards or the Myrinet LANAI.

Overall, most of the new polling approaches work remarkably well. They are all far better then basic polling scheme. On average, 'heur-t' is more than 60% better then basic polling for the medium network. Average performance differences between 'heur-t' and 'ifast' range are less than 0.4% for all three networks. Differences increase slightly with the faster networks because overall running times decrease with network latency. Differences between 'loop-t' and 'ifast' range from 1.4% for 'slow' to 2.2% for 'fast'. The 'loop-t' approach might be preferred over 'heur-t' in practice because of its greater conservatism. 'Heur-t's speedup is averages 11% higher than 'loop' which puts polls in all loops indiscriminately. This gap is likely to increase with newer hardware.

A somewhat surprising result is that the average speedup with slow interrupts is within 5% of that of fast interrupts for all three network configurations. This implies that even slow interrupts provide good performance for most applications.

The good performance of the 'heur-t' scheme means that relatively large polling costs can be tolerated without undue impact on performance. A 100-cycle poll would add a maximum of 1% to overall running time because it is only called every 10000 cycles. Results in Section 4 show that even larger poll costs can be tolerated without adverse effects on performance. This is important because many interfaces can not be used with cheap inline 20-cycle polls. For example, an MPI poll on AIX 4.2 costs over 5 *m*secs, more than 300 cycles. It may be possible to check the status of the switch directly with access to source. However, any such hack will not be portable, obviating the main reason for using MPI in the first place.

The 50-microsecond polling watchdog performs within 1% of 'ifast' for all three networks. The one-millisecond watchdog averages between three and four percent slower than 'ifast'.

7. RELATED WORK

Neither multi-threading nor polling are new ideas. Multi-threading has been studied not only for masking request/reply latencies as we do, but also at the hardware level for masking cache misses [17]. In this work, we show how multi-threading compares in hiding the remote request latencies vs. fast software interrupts in reducing those latencies.

The sensitivity of communication mechanisms to message latency was studied by Chong [7]. The main issue studied here is how shared memory and message passing compare to each other with respect to network latency. For message passing, they agree with Brewer [6] that polling performs better when performance is dominated by communication. In addition to direct costs, interrupts can lead to significant load imbalance. However, none of their applications assume asynchronous message arrivals.

Brewer described the remote queues abstraction [5, 7] for the same environment. Remote queues are a fine-grained mechanism for integrating polling and interrupts. Message-passing overheads and tradeoffs on the Alewife are quite different than on clusters of workstations.

Martin et al. [19] examined the effects of latency, overhead, and bandwidth on parallel application performance. They found significant slowdowns as the communication overheads and minimum message gap increase, while latency and bandwidth are the most important factor for request/reply applications. Their work assumed polling.

Mukherjee et al. [21] propose putting the network interface on the memory bus, which would enable the processor to cache status, control, and data network interface registers. This would provide very cheap polling through access to the cache, combined with speculative execution. The polling would then be cheap enough to be used unconditionally in all loop iterations, and eliminate the majority of notification delay. One drawback of this approach is that it could only be widely implemented if memory bus interfaces are standardized. Nonetheless, this approach should perform better then the fast interrupts that we simulate, and far better then basic polling mechanism used today, i.e., polling after message sends.

Falsafi et al [10] studied the scheduling of protocol operations on SMP nodes. They show that using a dedicated processor within an SMP is cost effective when the protocol is light-weight, there are more then two processors per SMP node, or applications are communication intensive. Dedicated processors poll constantly, so notification delay is only dependent on the occupancy of the dedicated processor. Our work differs in that it is directed at heavy-weight protocols and single-processor systems. However, the use of dedicated processors is certainly a viable approach for medium to large-scale multiprocessors. On smaller machines, the loss of a processor for computation might make the cost of this approach prohibitive.

Our work on polling is most similar to the Polling Watchdog study [18]. Our simulation of the Polling Watchdog shows it is nearly as effective in our domain as with bulk-synchronous applications. We showed that the reason for this effectiveness is the extremely long tail of the distribution in delay lengths. Our study shows that polling and multi-threading are at least as effective, without requiring hardware assistance.

8. CONCLUSIONS

Advances in current network technology are finally catching up with those in processor technology. Unfortunately, many new high-performance network interfaces require explicit polling in order to detect incoming messages or requests. The most common polling scheme consists of a poll after each message send. We show that this scheme results in inordinately large notification delays and poor performance for many applications.

Note that we are not arguing that our polling schemes or multi-threading should be used instead of interrupts. Instead, the focus of this paper is on whether the lack of interrupts and hardware-based notification mechanisms dooms many applications to poor performance. Happily, this is not the case.

We explore two strategies for dealing with notification delay. Multi-threading can be used to tolerate notification delay. We show that using four threads per node increases average performance for our applications from 62% of the best possible case (fast software interrupts) to 86%.

Second, we evaluate several strategies for decreasing notification delay directly by automatically inserting polls. We show that the cost of polling can be effectively amortized by using an optimized check sequence. The resulting applications perform within 1% of (very) fast interrupts. Furthermore, we show that this amortization can be applied to relatively expensive polling operations, allowing the use of standard polling interfaces. Hence, the lack of explicit interrupt support in current network interfaces does not preclude good performance for our application domain.

Finally, we investigated the distribution of notification delays in our applications. We found that the majority of delay is caused by a relatively small number of large delays. Eliminating these delays through a high-threshold timeout scheme, such as the one-millisecond polling watchdog, allows the applications to perform within a few percent of a system that supports fast interrupts.

DSM applications are currently a niche within the small niche of parallel applications. However, the advent of Java, CORBA, and other new distributed object technologies can only cause the importance of reducing notification delay to increase. Furthermore, our findings bode well for the use of non-dedicated networks of workstations to run distributed applications, and complement recent work on emergent co-scheduling [25].

9. References

[1] "Virtual Interface Architecture Specification," http://www.viarch.org, 1997.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," in *Proceedings of the '93 CompCon Conference*, February 1993.

[3] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro*, vol. 15, pp. 29-36, 1995.

[5] E. A. Brewer, F. T. Chong, L. T. Liu, S. Sharma, and J. Kubiatowicz, "Remote Queues: Exposing Message Queues for Optimization and Atomicity," in *ACM Symposium on Parallel Algorithms and Architectures*, 1995.

[6] E. A. Brewer and B. C. Kuszmaul, "How to Get Good Performance from the CM-5 Data Network," in *International Parallel Processing Symposium*, 1994.

[7] F. Chong, R. Barua, F. Dahlgren, J. Kubiatowicz, and A. Agarwal, "Sensitivity of Communication Mechanisms to Bandwidth and Latency," in *Proceedings of 4th Int'l Symposium on High Performance Computer Architecture (HPCA)*, February 1998.

[8] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[9] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. H. J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," in *Supercomputing 94*, 1994.

[10] B. Falsafi and D. A. Wood, "Scheduling Communication on an SMP Node Parallel Machine," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, February 1997.

[11] E. W. Felton and D. McNamee, "Improving the performance of message-passing applications by multithreading," in *Proceedings of the Scalable High Performance Computing Conference*, April 1992.

[12] R. Gillett, "Memory Channel Network for PCI," *IEEE Micro*, vol. 16, pp. 12-18, 1996.

[13] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[14] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[15] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the 1994 Winter Usenix Conference*, January 1994.

[16] Lee, Crowley, Baer, Anderson, and Bershad, "Execution Characteristics of Desktop Applications on Windows NT," in *The 25th Annual International Symposium on Computer Architecture*, June 1998.

[17] B.-H. Lim and R. Bianchini, "Limits on the Performance Benefits of Multithreading and Prefetching," in *Proceedings of the International Conference on the Measurement and Modeling of Computer Systems*, 1996.

[18] O. Maquelin, G. R. Gao, H. H. J. Hum, K. Theobald, and X. Tian, "Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling," in *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, May 1996.

[19] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," in *Proceedings of the 1997 International Symposium on Computer Architecture*, June 1997.

[20] T. C. Mowry, C. Q. C. Chan, and A. K. W. Lo, "Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[21] S. S. Mukherjee and M. D. Hill, "A Case for Making Network Interfaces Less Peripheral," in *Hot Interconnects V*, August 1997.

[22] L. Prylli and B. Tourancheau, "BIP: a new protocol designed for high performance networking on myrinet," in *Workshop PC-NOW, IPPS/SPDP98*, 1998.

[23] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing TLB and Memory Overhead Using Online Superpage Promotion.," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[24] D. Scales and K. Gharachorloo, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory," in *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[25] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien, "Dynamic coscheduling on workstation clusters," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[26] A. Srivastava and A. Eustace, "ATOM: A system for Building Customized Program Analysis Tools," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, May 1994.

[27] C. Thekkath and H. Levy, "Hardware and Software Support for Efficient Exception Handling," in *Proceedings of the 6th International Conferance on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[28] K. Thitikamol and P. Keleher, "Multi-Threading and Remote Latency in Software DSMs," in *The 17th International Conference on Distributed Computing Systems*, May 1997.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.