Adaptive Receiver Notification for Non-Dedicated Workstation Clusters*

Michael Schmitt   Max Ibel   Anurag Acharya   Klaus Schauser

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{*schmittm, ibel, acha, schauser*}@*cs.ucsb.edu*
Tel: (805) 893-3996
FAX: (805) 893-8553

TRCS98-14

May 4, 1998

Abstract

Efficient communication in a NOW environment can be a challenging task. Depending on the application, the architecture of the nodes and the characteristics of other processes running on the nodes, different communication mechanisms can be appropriate. In this paper, we evaluate an adaptive strategy which provides multiple communication mechanisms and selects between them depending on the current situation. We focus on strategies for receiver notification under varying load conditions. Previous research has shown that in a non-dedicated environment, spin-block strategies can avoid scheduling anomalies which can cause several orders of magnitude of performance degradation. However, research on these strategies have focused on fixed spin-delay. We investigate an adaptive strategy that uses run-time information to select the spin-delay in an online manner. Our strategy achieves better performance than static spin-block strategies on workstation clusters with varying load.

1 Introduction

Networks of workstations (NOWs) are a popular computing platform as they provide good performance at a reasonable cost and can be built in an incremental manner. Individual nodes in a NOW are independent machines and run a standard operating system. Efficient communication in a NOW environment, however, can be a challenging task if the applications running on the NOW are irregular in terms of load and communication patterns or, as is often the case, the NOW is not dedicated to a single parallel application. One's

*This research was partially supported by a grant from Intel Corporation and by a grant from the California Micro program.

task can be even harder if the incremental growth of the NOW results in a variety of architectures, operating systems, clock-speeds and so on.

Depending on the application, the architecture of the nodes and the characteristics of other processes running on the nodes, different communication mechanisms can be appropriate. We evaluate an adaptive framework which provides several communication mechanisms and selects between them depending on the current situation. Such a methodology can offer significant performance improvements over static communication schemes in environments with large variations in computation and communication requirements.

We focus on strategies for receiver notification in a message-passing environment. Efficient receiver notification is a well-known problem for user-level communication schemes. On modern architectures, polling requires just a few reads from memory, whereas interrupts and context switches for asynchronous notification are much more expensive. As a result, a *spin-only* strategy, which requires the receiver to spin in a loop until the message arrives, provides superior performance in dedicated environments. In non-dedicated environments, however, spin-only can perform poorly since each node is independently scheduled and a sender-receiver pair could be scheduled in different time-slices such that the latency of messages jumps from a few μs to as long the scheduling quantum – tens to hundreds of ms, as illustrated in Figure 1. This problem becomes worse as the load on the NOW increases. In a non-dedicated setting, misaligned scheduling not only slows down the parallel application, but also wastes time in polling which could be gainfully utilized by other processes.

Implicit co-scheduling [1, 2, 3, 4] has been proposed as a way to improve the scheduling coordination without modifying the operating system. It takes advantage of the communication performed by the parallel application to keep communicating peers (roughly) in sync. A straightforward way to do this is to require the receiver to block when it waits for a message. When the sender sends the message, it wakes up the receiver (usually by generating an interrupt). We refer to this as the *block-only* strategy. Block-only behaves better than spin-only on a non-dedicated cluster, but suffers from the high cost of interrupts and context-switches. A hybrid strategy amortizes these costs by requiring the receiver to spin for a given time-period (*spin-delay*) and to block thereafter. This mechanism, first described and proven 2-competitive in [5], is referred to as *spin-block*, and has been successfully used in [2, 3, 4]. Figure 1 sketches the spin-only and spin-block protocol.[1]

However, research on *spin-block* strategies has focused on fixed spin-delays. An examination of the performance of parallel applications in the context of a varying sequential workload indicates that even though the sequential load on a NOW node tends to be low (less than one), it is often bursty and irregular. Our results show that for such workloads, the optimal spin-delay varies with the granularity of the parallel application and the system load. We investigate an adaptive strategy that uses run-time information to select the spin-delay in an online manner.

Since workstation clusters are likely to be heterogeneous, portability is an explicit goal. Accordingly, we focus on techniques which can be implemented at the level of communication libraries and which require no changes in the operating system or the network interface.

Contributions

- We describe a portable spin-block implementation that does not require changes to the operating system or the network interface.

[1] In multi-threaded environments, we have the additional choice of yielding instead of blocking. In this paper we are only looking at single-threaded applications.

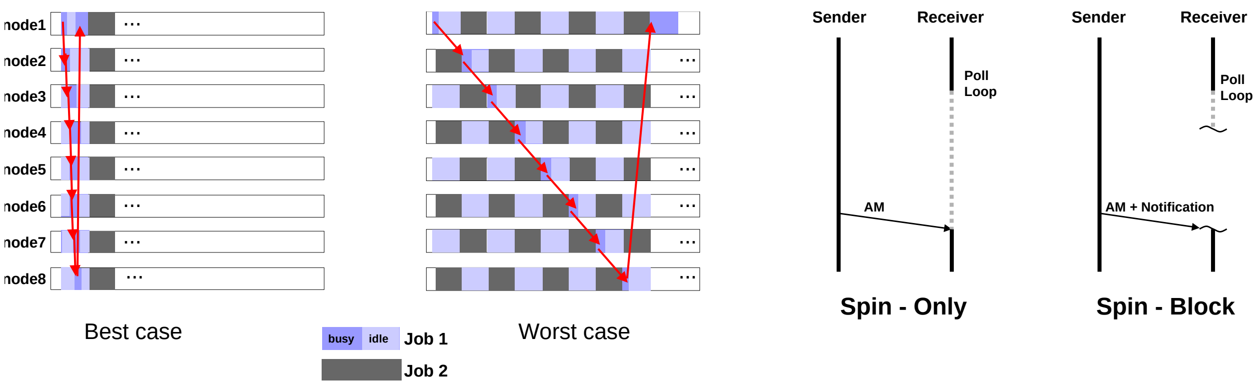

Figure 1: *Left: Best case (latency determined by network) and worst case (latency determined by quantum size) for scheduling a parallel application that sends a token along all nodes in a ring on a cluster. Right: Implementation of spin-only vs spin-block.*

- We present two adaptive algorithms that use run-time information to select the spin-delay.
- We evaluate our algorithms and our implementation under varying load conditions. Our adaptive algorithms achieve significant performance improvements over static spin-block strategies.

The rest of this paper is organized as follows. In Section 2, we describe our experimental setup and our Sun Ultra workstation cluster. In Section 3, we introduce our implementations of the alternative strategies for receiver notification and present results from evaluating their performance on our cluster. We present the algorithms for selecting the spin-delay in Section 4. We also evaluate their performance under varying load. We compare our approach to related work in Section 5. Finally, we conclude with a summary of our results and a discussion of future work.

2 Experimental Testbed

Our testbed consists of two clusters of Sparc-based workstations. In this study we primarily use a 4-node cluster consisting of 250 MHz PCI-based Ultra-30 workstations running Solaris 2.6. Our setup is shown in Figure 2.

The interconnection network for our cluster is the *Scalable Coherent Interface* (SCI) [6]. Like Memory Channel [7], SCI provides a limited form of (non-coherent) shared memory. A node can export a memory segment to other nodes, which can then directly read and write to it – the network adapters translate accesses to remote memory unconditionally with no caching into network transactions. The operating system is only involved in setting up memory segments and thus securing process integrity. After setup, simple loads and stores are sufficient. This allows low latency, high-bandwidth communication ($3\mu s$ latency for one-word stores and bandwidths of 66 MB/s on our cluster). Unlike many user-level communication architectures, SCI completely virtualizes the network interface; multiple processes can concurrently and transparently use the network interface. For detailed performance numbers on our cluster, refer to [8].

We use an **Active Messages**-based [9] communication library. We chose this approach as it uses simple and efficient protocols, yet is flexible enough to implement higher-level message-passing abstractions. An Active Message contains the address of a handler function which is executed on a receiving processor.

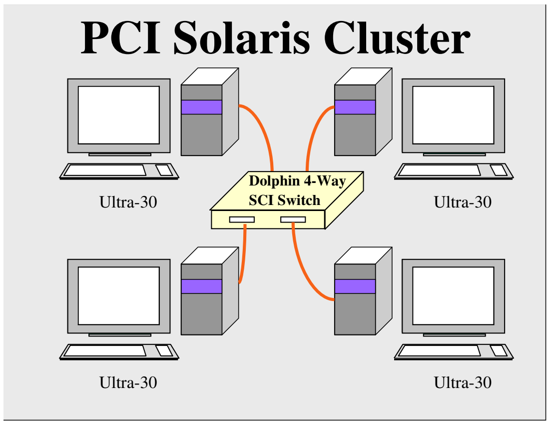

Figure 2: *Our PCI-based SCI cluster consist of four Ultra-30 workstations.*

Handlers are nonblocking, execute quickly, and may issue further communication only in a restricted way to simplify deadlock considerations. Active Messages often achieve significant performance improvements over more traditional communication mechanisms, and are ideally suited for high performance clusters such as ours. Note that receiver notification is one of the major challenges for Active Message implementations.

3 Implementation of Spin-Block on SCI

Our main goal for the implementation was to create a spin-block version of Active Messages that does not require modifications to the network interface (e.g. firmware, driver) or to the operating system (e.g. scheduling algorithm). We decided to use TCP sockets for notifications.[2] Using sockets, a receiver can just block on a read and incoming data is dispatched by the OS, unblocking the receiver. A less portable alternative would be to modify the SCI driver to generate interrupts. While it does provide a small performance improvement (generating and dispatching an interrupt using SCI takes $150\mu s$, whereas a TCP-band notification takes $225\mu s$), we believe the advantage would not outweigh the loss of portability. TCP also has the potential advantage of allowing the receiver to combine multiple incoming messages into a single notification.

Our library retains the Active Messages interface and can be used as a drop-in replacement for the standard library for the implementations of higher abstractions like Split-C [10], Titanium [11], MPI [12]). Since our implementation does not modify the driver or the firmware, it allows SCI to maintain virtualization of the network interface. This allows our implementation to be used for multiple concurrent application on the same set of nodes.

The receipt of a notification causes the OS to change the receiver's process state from blocked to runnable and to issue a scheduling decision. Since the process was blocked, its priority level is raised (it gets a "priority boost"). This priority boost causes the scheduling event to be most likely in favor of the parallel job; therefore it is essential to achieve implicit co-scheduling. We optimize our implementation of spin-block using a combination of sender and receiver time-stamps. A sender eliminates the notification if the last communication with the destination receiver happened within the spin-delay period.

[2]Currently we are using TCP over Ethernet. Since this might cause conflicts with other user workload using Ethernet we will move to TCP over SCI in future versions.

3.1 Evaluation

To evaluate our implementation of spin-block, we have developed a micro-benchmark called `simplebench` which imitates the operation of regular and irregular applications on our cluster. Each iteration of its main loop consists of a computation phase and a communication phase. The compute time for each iteration is chosen independently from a normal distribution (with $\sigma = 0.2 * mean$). The communication step in each iteration consists of 15 bulk operations which read or write a 64-byte chunk from a randomly selected peer. The benchmark is self scaling in that it continues until the 90% confidence interval of the execution time per iteration is within 10% of the mean. The main loop is executed at least 30 times to ensure that the sample is large enough.

To provide a repeatable environment for our micro-benchmark, we use an unloaded cluster. To generate the sequential load, we use one or more threads that execute an infinite loop (`for(;;);`). This creates a uniform load (of one or two) on each node. We varied the mean compute time per iteration between $100\mu s$ and $100ms$, and the spin-delay between 0 (equivalent to block-only) and $1024ms$ (equivalent to spin-only for the set of compute granularities used).

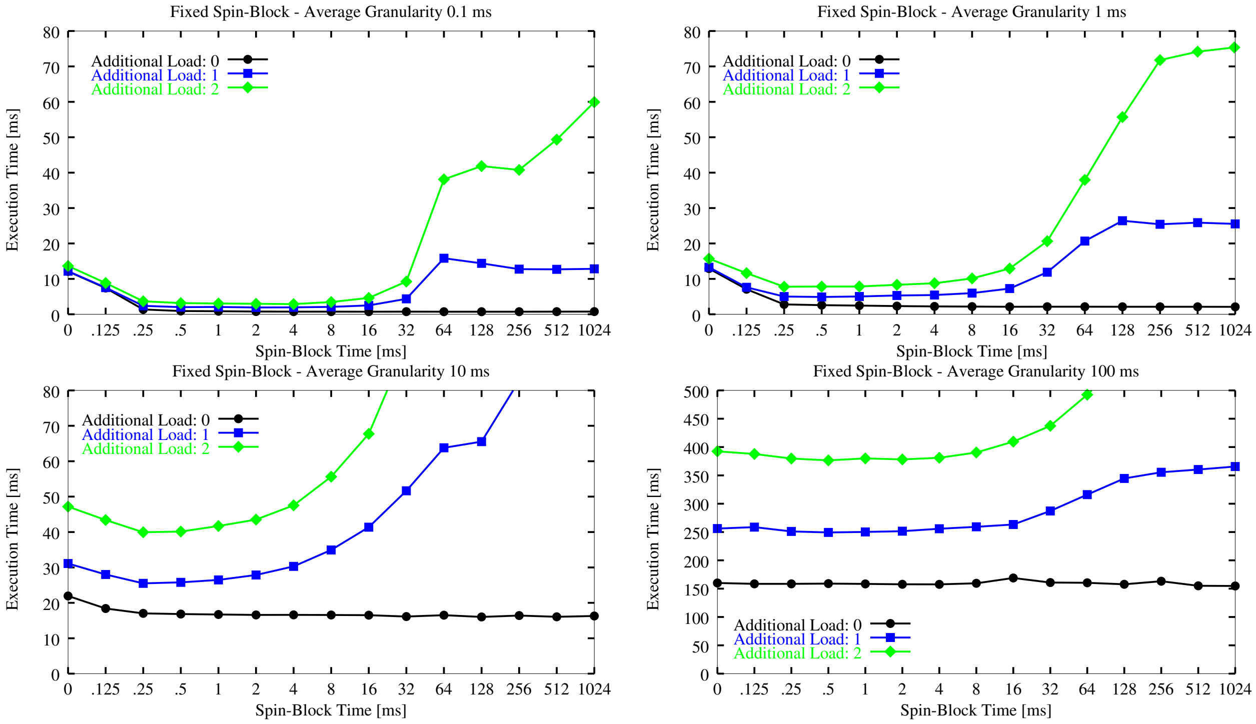

Figure 3: *Fixed spin-block: execution times of* `simplebench` *on four nodes with different computation granularities and additional workloads. The y-axis shows the time per iteration, the x-axis the spin-delay*

The results are presented in Figure 3. Ideally, the time per iteration (TPI) would be equal to the sum of the compute time and the time to deliver the messages. In practice however, TPI is always larger since the processes are not synchronized and may enter the communication phase later than their peers.

By looking at the graph for $100\mu s$ granularity we can see that on an unloaded cluster, the TPI starts at around $12ms$. As the spin-delay grows, the TPI drops to less than $1ms$. This shows that blocking is too expensive to use it unconditionally. However, increasing the spin-delay beyond $1ms$ only yields a small improvement. Nevertheless, spin-only is clearly the best strategy in a dedicated cluster. On a loaded cluster the picture changes: If the spin-delay is $\leq 125\mu s$, the TPI is dominated by the blocking overhead. For delays between $250\mu s$ and $8ms$, the TPI stabilizes to a value mainly depending on the load of the cluster. With a load of one the TPI is approximately twice that on an unloaded cluster, with a load of two, it is thrice as large. This indicates that the parallel application obtained the expected share of the CPU time proportional to the other load on the cluster. For spin-delays above $8ms$, the TPI increases dramatically by up to two orders of magnitude as load is increased because co-scheduling cannot be achieved any more in these cases. For larger compute granularities, the TPI for load 1 is actually *less* than twice the TPI for the dedicated case. This is because the sequential load can use the "spare" CPU time that could not be used in the dedicated case.

The results also indicate the importance of the priority boost obtained by blocking. As Figure 3 shows, TPI remains relatively constant in the range from $0.5\mu s$ to $8\mu s$. For larger spin-delays, the application is likely to be preempted before it reaches its blocking point (the scheduling quantum for high-priority jobs in Solaris is $20ms$). Consequently, the application is unable to acquire the priority boost which interrupts co-scheduling. In the case of larger compute granularities the range of almost-optimal spin-delays expectedly becomes smaller since the quantum size is reached earlier for a complete computation + communication step. These considerations are important for calibrating the adaptive schemes in the next section.

4 Adaptive spin-block mechanism

Our adaptive spin-block mechanism consists of three parts: first, a monitor that gathers status information; second, a strategy that selects the spin-delay period based on this information and third, a mechanism to switch-over from the previous spin-delay to the new one. It is important to keep the cost of monitoring down yet be able to gather sufficient information to make effective decisions.

We propose that a separate copy of the monitor runs on every node and samples readily available information such as CPU usage and number of transitions to the blocking state at regular intervals. This information is passed to a central node which makes the spin-delay selection at regular intervals. The new spin-delay is propagated to all nodes using a barrier to ensure an atomic switch-over. We present two algorithms: One that uses only CPU usage information to switch between spin-only and a fixed spin-delay and one that uses CPU usage and informations about the number of times the process blocks to select any spin-delay within $64\mu s$ and $64ms$.

Our basic adaptive protocol can be described in three steps:

1. All nodes report their locally collected system parameters once per monitoring interval to a designated root node by using out-of-band signaling, i.e. special remote-mapped memory locations.

2. The root node decides once per monitoring interval whether the current communication scheme should be changed/adjusted or not, depending on the data gathered. Note that the decision process at the root node and the monitoring points at every node are due to performance considerations not synchronized, but the age of monitoring data on the root node is at most one monitoring interval.

3. When the root node decides to change/adjust the communication scheme it instructs every other node to perform a cluster-wide asynchronous barrier to propagate the information about the change. As part of that barrier, every node adjusts its communication scheme.

	From Spin-Only	**From Spin-Block**
To Spin-Only	—	$3 - 20ms$
To Spin-Block	$50 - 200ms$	$3 - 20ms$

Table 1: *Overhead of spin-delay adjustment for different cases of current and new spin-delays. The granularity was chosen to be 10 ms.*

4.1 Monitoring and Basic Spin-Delay Adjustment Scheme

Our monitoring design currently determines the following system parameters at regular intervals:

1. `total_utime`: overall CPU time spent in user code (including all processes, excluding the kernel).
2. `parallel_utime`: CPU time spent in user code of the parallel job.
3. `block_calls`: number of blocking system calls. Assuming I/O calls are far fewer than communication events, this should represent the number of times we switch from spinning to blocking.

By comparing `total_utime` and `parallel_utime` for a node, we can determine its load in the last interval. Our first adaptive scheme, Bimodal Adaptation, just uses this information. Our second adaptive scheme, Multi-modal Adaptation, also considers the `block_calls` parameter to implicitly derive the granularity of the parallel job in addition to the load. Note that although our setup is based on Solaris 2.6, most operating systems (almost all Unix systems and newer Windows flavors) allow user access to these or very similar parameters.

The monitoring overhead to determine `total_utime`, `parallel_utime` and `block_calls` is about $50\mu s$ per sample on our platform. The decision process has negligible overhead. The cost of the barrier for propagating a change in the spin-delay depends highly on the network architecture, the cluster load and the application. Since our benchmark application does not poll while in a compute phase, a barrier might take as much as the longest compute phase, which also depends on the system load. Table 1 shows the three basic cases. The highest overhead ($50 - 200ms$) occurs during transition from spin-only (spin-delay is infinite or at least very high) to spin-block. This happens when at least one node changes from a dedicated to a non-dedicated state when previously all nodes have been in a dedicated state. Since the spin-only protocol is now inappropriate and prevents the application from reaching co-scheduling, the latency of the necessary barrier in our protocol is in the range of the length of the quantum. The transition from spin-block to spin-only and spin-block to spin-block (only in the multi-modal version) is less costly ($3 - 20ms$) since the current spin-block protocol does not incur much overhead and provides reasonable co-scheduling even if the current spin-delay is inappropriate to the new system state.

4.2 Bimodal Adaptation

The bimodal adaptation strategy uses load information from all nodes to make its decisions. As long as the load is low on all nodes, it concludes the cluster is dedicated and uses a spin-only protocol. When it detects significant load on any two nodes, its switches to a spin-block protocol. Thus, the choice of the protocol depends on the load on the two most heavily loaded nodes. Note that it is not beneficial to switch to spin-block when only one node in the cluster is non-dedicated. In such a situation, all other nodes have to wait anyway but do not gain anything by blocking. In fact, blocking is likely to yield worse performance due to the high overhead of receiver notification. To implement bimodal adaptation, we had to make two main decisions: (1) when should a node be considered non-dedicated, and (2) what spin-delay should be used for a non-dedicated cluster. We mark a node as non-dedicated iff `total_utime` $< 1.2 *$ `parallel_utime` on that node, that is, if the local processes receive at least 20% CPU time received by the parallel application. We determined this threshold empirically. Lower thresholds resulted in a large number of unnecessary transitions due to the daemon processes; higher thresholds resulted in undesirable blocking. We chose $4\ ms$ as the spin-delay for a non-dedicated cluster. This period was found to yield good results for light to medium loads which we expect to be the common case.

4.3 Multimodal Adaptation

The bimodal strategy uses only load information and ignores the impact of the priority boost received whenever a process blocks. Figure 3 shows the choice of the best spin-delay depends both on the load and the granularity of computation between successive communication phases. For larger computation granularities, the parallel application blocks less frequently and as a result, receives priority boosts less frequently. Consequently, it is less likely to be run as soon as a message arrives for it.

The multimodal strategy uses `block_calls` as a measure of the priority boost received by the parallel application. It tries to keep the number for `block_calls` within a desirable range – a very large number of `block_calls` indicates that the application is paying too much overhead for notification whereas a very small number for `block_calls` indicates that the application is not receiving sufficient priority boost needed to maintain implicit co-scheduling.

We determined empirically that 30-50 `block_calls` per second provide a good trade-off. The upper end of this range is derived from the 20 ms time-slice used by Solaris 2.6 for its highest priority processes. This would be achieved by an application that blocks just before each preemption. The lower end of this range is derived from empirical evaluation. It remains to be seen to what extent this range depends on the environmental factors, such as number of nodes, processor speed, network speed, protocol overhead, etc.

Multimodal adaptation uses the same scheme as bimodal adaptation to check the cluster is dedicated. If it is, it uses a spin-only protocol. If it is not, it adjusts the spin-delay to try to keep the number of `block_calls` in the 30-50 range. If at least one node has less than 30 calls, it decreases the spin-delay; if at least one node has more than 50 calls, it increases the spin-delay.

Figure 4 illustration the operation of both bimodal and multimodal adaptation and compares it with that of a hypothetical optimal adaptation scheme. Note that the multimodal scheme adjusts the spin-delay in (relatively) small increments and may take several steps to reach a stable value.

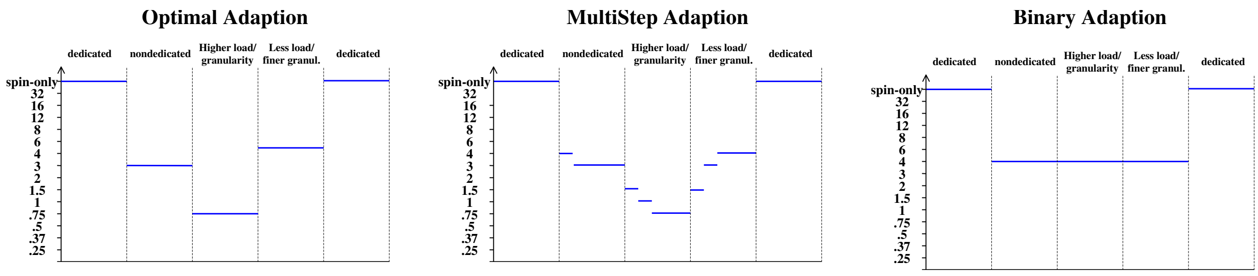

Figure 4: *Behavior of adaptive protocols for varying workload. The left graphs shows how a hypothetical optimal algorithm would behave. The other two graphs show our adaptive implementations.*

	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6
Load 0	100%	80%	60%	40%	20%	0%
Load 1	0%	15%	30%	45%	60%	75%
Load 2	0%	5%	10%	15%	20%	25%
Idle time	100%	40%	15%	5%	0%	0%

Table 2: *Load patterns for the comparison of the fixed and adaptive spin-block implementations. Idle time denotes the probability of all 4 nodes being idle at the same time (rounded to the nearest multiple of 5%).*

4.4 Experimental Evaluation

To evaluate the adaptive schemes, we developed a simple tool to generate a variable load on a cluster. This tool, which we refer to as `hogtool`, allows us to create a variety of load conditions. The `hogtool` has two parameters - a burst length and a probability mass function used to determine the load for individual bursts. In each burst, the probability mass function is used to randomly select the load for the duration of the burst. An exemplar probability mass function that could indicate no load for 80% of the bursts, load 1 for 15% of the bursts and load 2 for 5% of the bursts. It uses a pool of worker threads to implement the desired load. A separate copy of the `hogtool` runs on every node. Figure 5 shows a sample load that could be generated.

For our experiments we used bursts of length 10 seconds and the probability mass function shown in Table 2. For example, in load pattern 1 represents a dedicated cluster, whereas load pattern 6 represents a heavily loaded cluster (75% load 1, 25 % load 2 independently on every node). Patterns 2 through 5 lie in between these extremes.

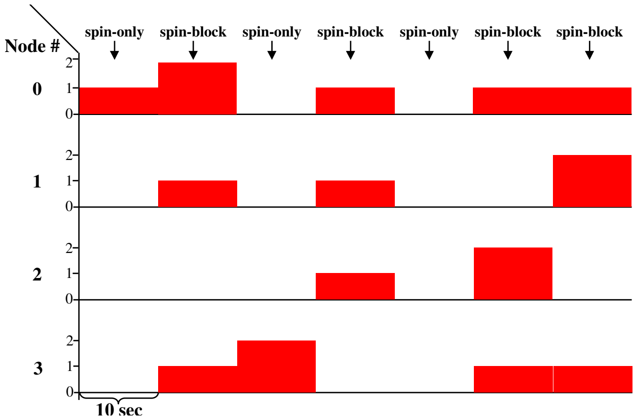

Figure 5: *A sample workload and the actions taken by the adaptive protocols.*

For our experiments, we used the `hogtool` (with the probability mass functions from Table 2) to provide the sequential workload and `simplebench` to provide the parallel workload. Figure 6 compares the the performance of the adaptive spin-block schemes with that of: (1) the spin-only scheme, and (2) the fixed spin-delay scheme with a range of spin-delays (500μs, $1ms$, $2ms$ and $4ms$). We selected these spin-delays based on the results presented in Figure 3. Execution times for all versions are normalized with respect to the execution time for the spin-only version on a dedicated cluster.

Our results indicate that an adaptive spin-delay clearly outperforms fixed spin-delays in the presence of

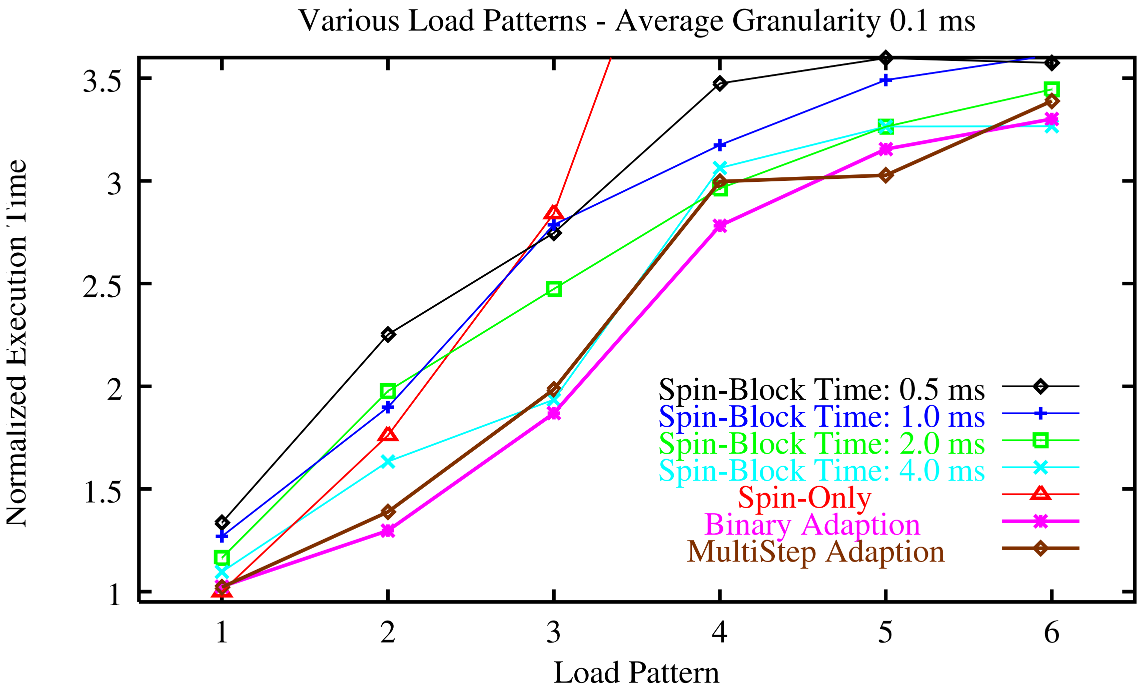

Figure 6: *Adaptive spin-block: execution times of* `simplebench` *on four nodes with* $100\mu s$ *granularity and load patterns for the additional workload. The* y*-axis shows the time per iteration, the* x*-axis the load patterns.*

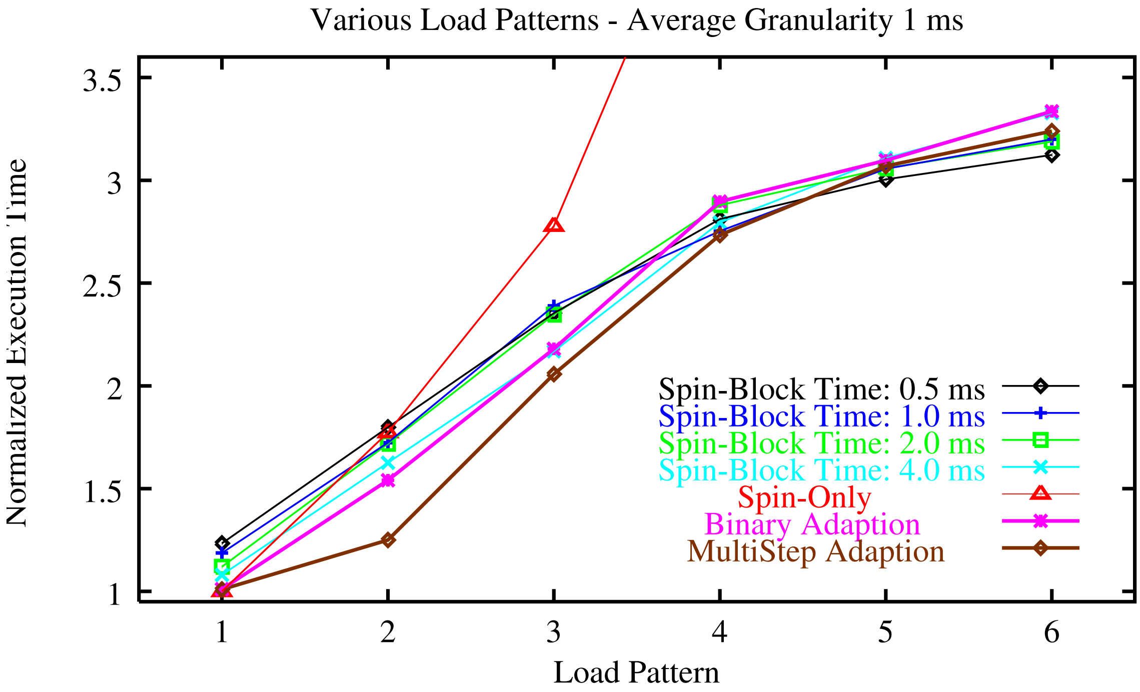

Figure 7: *Adaptive spin-block: execution times of* `simplebench` *on four nodes with* $1ms$ *granularities.*

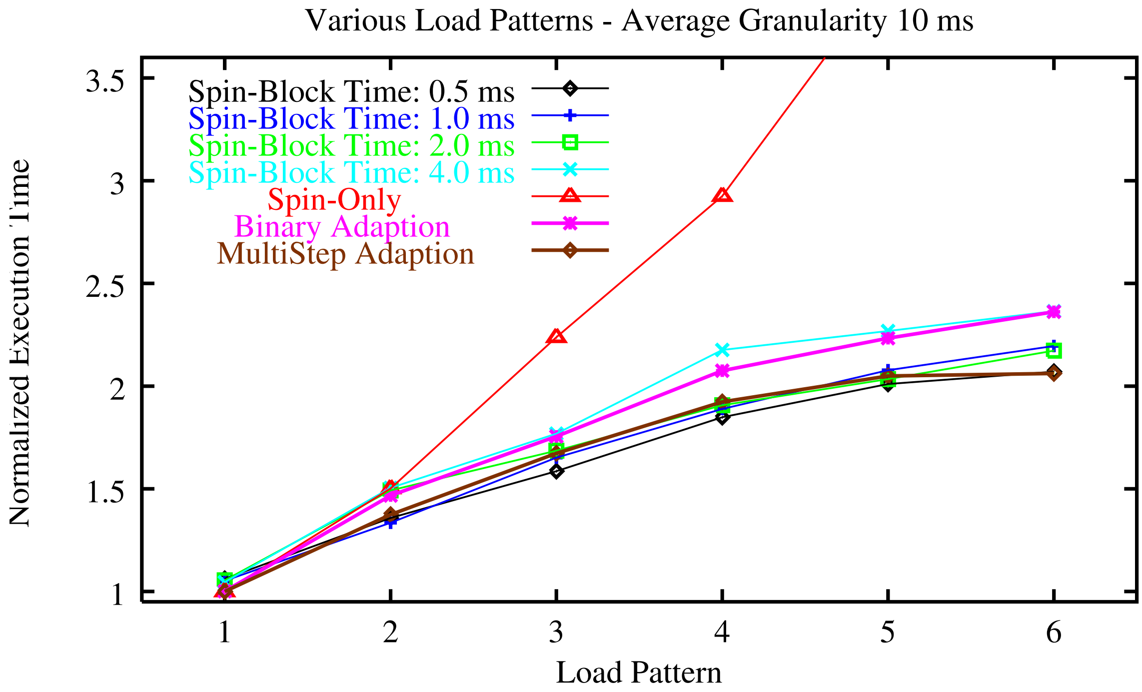

Figure 8: *Adaptive spin-block: execution times of* `simplebench` *on four nodes with* $10ms$ *granularities.*

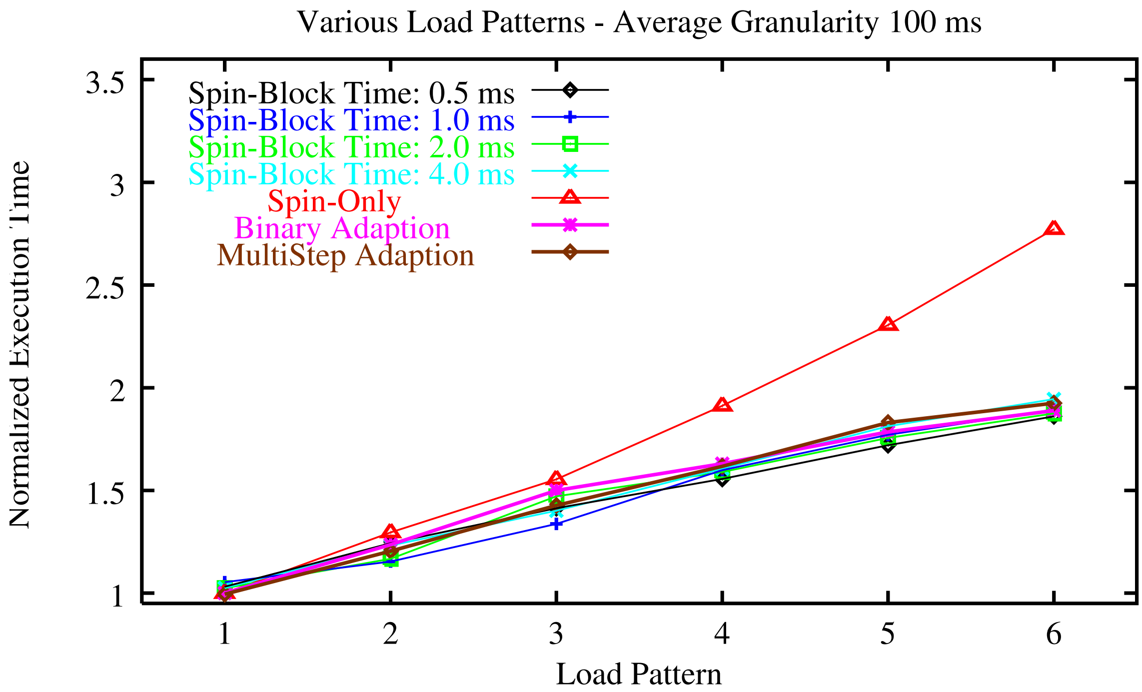

Figure 9: *Adaptive spin-block: execution times of* `simplebench` *on four nodes with* $100ms$ *granularities.*

variable load. While a fixed spin-delay is able to achieve good performance for a specific load condition, the adaptive schemes are able to mold to a variety of load conditions and are able to automatically converge to the appropriate spin-delay.

As the load increases, bimodal adaptation switches to the $4ms$ spin-delay. As long as the load is light to medium, this achieves good performance. Multimodal adaptation take advantage of its greater flexibility to converge to the *best* spin-delay for every granularity and load. For the smaller computation granularities (0.1 ms and 1 ms), a 4 ms spin-delay is the best fixed strategy, whereas for 10 and 100ms granularities, a 0.5 ms spin-delay is the best fixed strategy.

The adaptive schemes outperform most fixed schemes for the smaller computation granularities ($100\mu s$, $1ms$). For example, with a medium load (load pattern 3), bimodal adaptation performs more than twice as good as a fixed spin-delay of 0.5ms. For larger granularities, the optimal spin-delay is much smaller than 4 ms. As a result, the bimodal approach achieves the same performance as a fixed spin-delay of 4 ms. Multimodal adaptation chooses the correct spin-delay but since the communication is less frequent, the higher overhead of this version is not always completely amortized. Nevertheless, both adaptive protocols are within 5% of the optimal fixed spin-delay for every load and granularity.

To summarize, if the cluster is guaranteed to be dedicated, spin-only is best choice; if the cluster could be non-dedicated during the execution of a parallel application, an adaptive spin-delay is preferable – particularly if the load on the cluster and/or the granularity of computation in the parallel application can vary during its execution.

5 Related work

Adaptive communication strategies have been investigated by several researchers. Schmidt et al. [13] describe the ADAPTIVE Communication Environment (ACE), which is a framework for higher-level communication primitives used in distributed computing. ACE is based on C++, sockets and IPC. While ACE provide a comprehensive set of classes to facilitate distributed computing and is even available commercially, our goal was to provide an lightweight, low-latency message passing implementation suited for high-performance parallel computing.

Lim et al. [14] and Diniz et al. [15] studied adaptive locking protocols for parallel machines. Lim et al. propose a scheme that collects runtime information about lock contention and uses this information to select between a set of locking protocols for a distributed shared-memory machine. Diniz et al. combine runtime information with compile-time information regarding the commutativity of operations to selectively coarsen locks. These studies restrict themselves to the behavior of a single parallel application running on dedicated hardware and have access to precise information such as lock contention. In contrast, we focus on external perturbations by uncorrelated user processes and have access only to approximate information like CPU load and number of preemptions for a process.

Scheduling anomalies is the major problem for fine-grained parallel applications running on a non-dedicated cluster.[3] Recent research on communication libraries for non-dedicated clusters has focused on implicit co-scheduling and spin-block protocols [4, 2, 3].

Dusseau et al. [2] investigate multiple bulk-synchronous parallel jobs with varying load-imbalance. Their work differs from ours in several ways. They propose the use of micro-benchmarks to determine the optimal spin-delay for a cluster. These micro-benchmarks are run on an unloaded cluster. The spin-delay

[3] See [16] for a survey on scheduling strategies for parallel machines and workstation clusters.

determined using this technique is used thereafter. Our results do not support their findings of an optimal spin-delay for a given cluster, independent of the cluster load. Their implementation modifies the network interface to generate notifications on the receiver side if the parallel process is in blocking mode. Finally, they investigate the performance of clusters running *multiple parallel applications* whereas we focus on the the situation with a single parallel application with sequential processes on individual nodes.

Sobalvarro et al. [3] use a similar approach with a fixed spin-delay. Like Dusseau et al. [2], their goal is to design schedulers and network interfaces to better support implicit co-scheduling. To achieve that, they need modifications to the network driver, the network interface firmware and to the operating system. In contrast, our goal is to build portable and adaptive communication schemes. Our approach requires no modification to the network driver or the network interface or the operating system.

Damianakis et al. [4] also use a fixed spin-delay and use the network interface on the receiver side to wake up blocking parallel jobs. Special hardware of their SHRIMP multicomputer allows them to enable and disable the fixed spin-delay protocol at memory page granularity. The portability of their approach is limited to the SHRIMP architecture.

6 Conclusions and Future Work

In this paper, we presented two adaptive schemes for receiver notification in non-dedicated workstation clusters. Both schemes are based on a spin-block protocol and use information collected at runtime to dynamically select an appropriate spin-delay. The first scheme is bimodal and uses load information to switch between spin-delays of 4 ms and ∞ ms. The second scheme extends this to select from a set of spin-delays. It estimates the granularity of computation in each iteration of the parallel application being executed and uses this estimate to guide its decision.

Our results indicate that an adaptive spin-delay clearly outperforms fixed spin-delays in the presence of variable load. While a fixed spin-delay is able to achieve good performance for a specific load condition, the adaptive schemes are able to mold to a variety of load conditions and are able to automatically converge to the appropriate spin-delay. If the cluster is guaranteed to be dedicated, spin-only is the best choice (as expected). However, if the cluster is likely to be non-dedicated during the execution of a parallel application, an adaptive spin-delay is preferable – particularly if the load on the cluster and/or the granularity of computation in the parallel application can vary during its execution. Of the two schemes presented in this paper, multimodal adaptation is superior as it is able to deal with a wider range of load conditions with very little additional overhead.

Our implementation of spin-block is based on explicit notification from the sender. While our approach is portable and lends itself naturally to new architectures like VIA, it is unclear if its sender-initiated nature could cause too many unnecessary notifications on larger clusters. We plan to study the performance of our schemes for larger clusters in the near future. We also plan to explore the use of asynchronous interrupts as well as more responsive and intelligent adaptive algorithms.

In the longer term, we plan to add additional communication mechanisms to our framework. We are currently exploring the use of the SCI memory-mapping primitives for implementing higher-level shared-memory abstractions. We would also like to evaluate the utility of program annotations to augment the information collected by runtime monitoring and to enable the runtime system to make better choices. Finally, we would like to port our framework to the Virtual Interface Architecture [17], which provides a similar interface and appears to be a likely candidate for commodity high-performance networking for workstation clusters.

References

[1] A. Dusseau and D. Culler. Extending proportional-share scheduling to a network of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, 6 1997.

[2] A. Dusseau, D. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *Sigmetrics'98 Conference on the Measurement and Modeling of Computer Systems*, 6 1998.

[3] P. Sobalvarro, S. Pakin, A. Chien, and W. Weihl. Dynamic coscheduling on workstation clusters. In *Proceedings of the International Parallel Processing Symposium*, 1998.

[4] S. Damianakis, Y. Chen, and E. Felten. Reducing waiting costs in user-level communication. In *11th Intl. Parallel Processing Symposium*, 4 1997.

[5] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for shared memory multiprocessors. In *Proc. 13th ACM Symposium on Operating Systems Principles*, 1991.

[6] IEEE, 345 East 47th Street, New York. *IEEE Standard for Scalable Coherent Interface (SCI)*, 1993.

[7] R. Gillet and R. Kaufmann. Experience using the First-Generation Memory Channel for PCI Network. In *Hot Interconnects IV — Symposium on High Performance Interconnects*, August, 96.

[8] Anonymous. Anonymous. In *Anonymous*, 1997.

[9] T. von Eicken. *Active Messages: an Efficient Communications Architecture for Multiprocessors*. PhD thesis, Computer Science Division — EECS, U.C. Berkeley, December 1993.

[10] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.

[11] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, February 1998.

[12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1.1 edition, June 1995. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[13] D. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, 1994.

[14] B. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 10 1994.

[15] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the SIGPLAN '97 Conference on Program Language Design and Implementation*, Las Vegas, NV, June 1997.

[16] Dror G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research report RC 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995.

[17] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Hot Interconnects V — Symposium on High Performance Interconnects*, August 97.