Coordinated Thread Scheduling for Workstation Clusters Under Windows NT

Matt Buchanan and Andrew A. Chien
Concurrent Systems Architecture Group
Department of Computer Science, University of Illinois

Coordinated Thread Scheduling for Workstation Clusters Under Windows NT

Matt Buchanan and Andrew A. Chien
Concurrent Systems Architecture Group
Department of Computer Science, University of Illinois
(`{mbbuchan,achien}@cs.uiuc.edu`)

Abstract

Coordinated thread scheduling is a critical factor in achieving good performance for tightly-coupled parallel jobs on workstation clusters. We are building a coordinated scheduling system that coexists with the Windows NT scheduler which both provides coordinated scheduling and can generalize to provide a wide range of resource abstractions. We describe the basic approach, called "demand-based coscheduling", and implementation in the context of Windows NT. We report preliminary performance data characterizing the effectiveness of our approach and describe benefits and limitations of our approach.

1. Introduction

Coordinated scheduling for parallel jobs across the nodes of a multiprocessor is well-known to produce benefits in both system and individual job efficiency [1, 5, 6]. Without coordinated scheduling, parallel jobs suffer high communication latencies between constituent threads due to context switching. This effect is exacerbated if the thread scheduling for individual nodes is done by independent timesharing schedulers. With high performance networks that achieve latencies in the range of tens of microseconds, the scheduling and context switching latency can increase communication latency by several orders of magnitude. For example, under Windows NT, CPU quanta vary from 20 ms to 120 ms [3], implying that uncoordinated scheduling can cause best-case latencies on the order of 10 μs to explode by three to four orders of magnitude, nullifying many benefits of fast communication. While multiprocessor systems typically address these problems with a mix of batch, gang, and timesharing scheduling (based on kernel scheduler changes), the problem is more difficult for workstation clusters in which stock operating systems kernels must be run. Coordinated scheduling reduces communication latencies by coscheduling threads that are communicating, thereby eliminating the context switch and scheduling latencies from the communication latency. Another important problem is that uncoordinated scheduling can reduce system efficiency as increased latency can decrease the efficiency of resource utilization. As high-performance networks and CPUs become more affordable for high-end computing, scheduling emerges as an important factor in overall system performance.

A low-latency, high-bandwidth messaging layer works to bridge the mix of interconnect, operating system, and user applications, delivering the raw performance of the interconnect to the software [14,2,8,9]. Such layers are essential because they make available the high performance of the underlying network hardware to applications. Illinois Fast Messages (FM) is such a messaging layer [14], and is a key part of the Concurrent System Architecture Group's High Performance Virtual Machines (HPVM) project [15], which seeks to leverage clusters of commodity workstations running Windows NT to run high-performance parallel and distributed applications. Fast Messages can deliver user-space to user-space communication latencies as low as 8 μs and overheads of a few μs. However, such levels of performance implies avoiding interrupts and system calls, so systems such as FM use polling to detect communication events, and therefore only delivers peak communication performane when effective coscheduling is achieved.

Coscheduling for clusters is a challenging problem because it must reconcile the demands of parallel and local computations, balancing parallel efficiency against local interactive response. Ideally a coscheduling system would provide the efficiency of a batch-scheduled system for parallel jobs and a private timesharing system for interactive users. In reality, the situation is much more complex, as we expect some parallel jobs to be interactive. Furthermore, in a cluster environment, kernel replacement is difficult at best, so we restrict ourselves to approaches that involve augmentation of existing operating system infrastructure.

Our approach to coordinated scheduling is Demand-based Coscheduling (DCS) [4, 7] which achieves coordination by observing the communication between threads. The essence of this approach is the observation that only those threads which are communicating need be coscheduled, and this admits a bottom-up, emergent scheduling approach. This approach can achieve coscheduling without changes to the core operating system scheduler and without changes to the applications programs. DCS causes the Windows NT scheduler to schedule communicating threads in a parallel job to run at roughly the same time, thereby minimizing communication latency.

Our implementation of DCS in Windows NT coexists with release binaries of the operating system require a customized device driver for the network card (in this case the Myrinet [13] card). This driver memory maps the network device into the user address space. The device driver, combined with special Myrinet firmware, influences the operating system scheduler's decisions by boosting thread priorities, based on communication traffic and system thread scheduling. The DCS algorithms are designed to drive the node OS schedulers into synchrony, achieving coscheduling among parallel threads that are closely communicating while simultaneously preserving fairness of CPU allocation. Our experiments evaluate an implementation of DCS for Windows NT, demonstrating that such an architecture is feasible, and validating DCS as a –promising approach for coscheduling. However, more extensive experiments are required before stronger conclusions can be drawn.

The remainder of this paper is organized as follows. Section 2 describes demand-based coscheduling briefly and our implementation approach for DCS. Section 3 describes performance with DCS, and section 4 presents some concluding remarks.

2. Demand-based Coscheduling

2.1. Overview

Demand-based coscheduling makes one central observation about the problem of scheduling parallel jobs, that only *communicating* threads need to be coscheduled to overcome scheduling and context switch latencies. DCS is driven directly by message arrivals: Whenever a message arrives, an opportunity for coscheduling arises. If no thread that can receive the message is currently running, DCS decides whether to preempt the current thread to synchronize the sending and receiving threads. The decision may be based on many factors, but in general DCS attempts to strike a balance between maximizing coscheduling performance and ensuring that the CPU is allocated fairly.

At the highest level, DCS has three key elements:

1. Monitoring communication and thread scheduling,

2. Deciding whether to preempt the currently running thread, and

3. Inducing the scheduler to select a particular thread.

The most direct approach to all three elements of DCS is to modify thread scheduler, embedded at the heart of the operating system kernel. However, because our goal is to support clusters running retail operating systems, such an approach has at least three drawbacks. First, modified kernels are unlikely to be run on a large number of systems, so such an approach will preclude large scale use of the coscheduling technology as middleware. Second, kernel modifications will tie the implementation to a particular operating system and version, increasing the software maintenance overhead, and making it difficult to leverage new operating systems features. The third and final drawback is proprietary concerns relating to source and binary distribution. An external implementation is generally more easily distributable.

DCS has been simulated extensively and implemented in the context of Solaris 2.4 [7,4]. The Solaris 2.4 implementation served as a model for and is similar in many ways to our Windows NT implementation

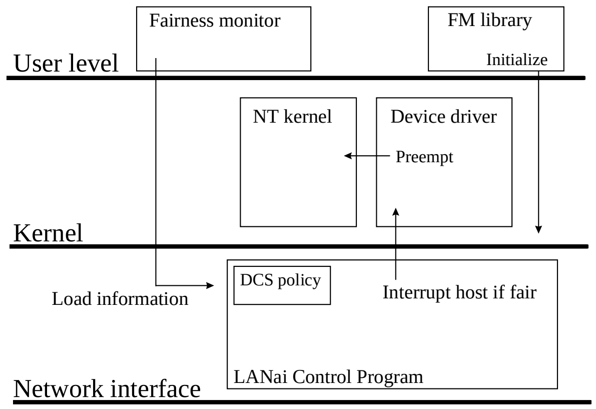

Figure 1. DCS organizational chart

2.2. Implementation

We implemented DCS for Windows NT 4.0 for the Intel *x*86 family of CPUs. Our implementation for Windows NT consists of four distinct parts: a DCS-aware Myrinet driver, the Fast Messages user-level library, a Fairness monitor, and DCS-aware FM firmware (a “LANai Control Program”, or LCP) that runs on Myrinet card. These elements interact as shown in Figure 1. The Fairness monitor in combination with the device driver monitors thread scheduling in the system, the modified firmware uses this information to decide whether to preempt the current thread, and the modified device driver induces the kernel scheduler to choose the desired thread for DCS.

2.2.1. Fairness Monitor

Our DCS implementation monitors thread and communication activity to ensure fairness of CPU allocation. The Fairness monitor runs as user-level (and thereby can access the NT kernel’s performance data for the length of the run queue). The average run queue length is written to the network card periodically, allowing the Myrinet firmware to moderate the frequency of priority boosts to ensure fair CPU allocations. Status information for the current thread is provided by the device driver as indicated below.

2.2.2. Myrinet Firmware

The DCS aware firmware is a modified version of the FM LANai Control Program and in addition to its basic communication function, the firmware makes preemption decisions based on the monitoring information provided by the Fairness monitor and the device driver.

Based on the run-queue length, current thread information, and the communication activity, the Firmware decides whether the current thread needs to be preempted (via an interrupt) and the device driver invoked to take DCS-related action. The decision procedure used by the Firmware is described below.

To determine whether a given thread is running, the Firmware periodically scans (approximately once per millisecond) the context switch information provided by the device driver. The LCP sets a flag if a communicating thread is running, and when a packet arrives, evaluates the following condition:

```
!threadIsRunning && fairToPreempt()
```

If the condition is true, then the LCP interrupts the host. The fairness criteria is critical and we adopt the approach taken in [4] to decide whether to interrupt the host. For a given thread, we interrupt if the following inequality is true:

$$\mathbf{2^E(T_C - T_P) + C \geq T_q R}$$

where

- T_C is the current time,
- T_P is the time the host was last interrupted to schedule this thread,
- T_q is the length of a CPU quantum (120 ms under Windows NT Server [3]),
- R is the number of threads waiting for the CPU,
- E and C are constants chosen to balance fairness and performance.

This approach limits the number of preemptions performed on behalf of a communicating thread by requiring that T_C - T_P, the time since the last preemption, is no less than the time it would take the CPU to service all of the ready threads if each thread ran for its entire quantum. The decay function 2^E and constant C afford some flexibility in tuning the inequality to allow for perturbations, such as those caused by threads that do not expire their quanta and priority-decay scheduling. The Firmware uses the Myrinet card’s on-board clock (0.5 μs granularity) to track T_C

and T_P. Under NT Server, the quantum size is constant.

Since the LCP scans the context switch information at one-millisecond intervals rather than for each packet, the information that `fairToPreempt()` uses may be stale. The scanning period involves a tradeoff between per packet overhead and staleness of the data. Since NT typically switches threads on tens of milliseconds time scale, we choose to reduce the per packet overhead.

2.2.3. Device Driver

We explored two basic approaches to the device driver, and describe both here as an illustration of what turned out to be difficult about manipulating the kernel scheduler to achieve the desired schedule. We term these two approaches **thread select** which makes use of a thread select callback, and **priority boost** which uses the thread priority boosting mechanism.

Thread Select Our initial implementation of DCS used NT's *thread select notify callback*, implemented in multiprocessor versions of the kernel. The scheduler passes the handle of a thread it proposes to select, and the callback returns a boolean value that it uses as a hint in deciding whether the given thread is appropriate to schedule. To cause the scheduler to favor a given thread when the thread has messages pending, the callback would simply reject the scheduler's choices until it proposed the preferred thread.

Unfortunately, the thread select notify callback is not suitable for DCS because its influence on thread scheduling decisions is limited. The scheduler uses several other criteria in addition to the callback's return value in choosing a thread to run, including the time the thread has been waiting for a CPU and its processor affinity [10]. Of course, if a competitor thread has a higher priority than a communicating thread, the scheduler may never propose a communicating thread to the callback. Thus, there is no guarantee that a communicating thread will be offered, much less scheduled at an appropriate time. Thus, using the callback to modify the scheduler's behavior was not a viable implementation for DCS.

Priority Boost This approach boosts the priority of a thread DCS would like to schedule which in general causes the NT scheduler to schedule the desired thread. However, since the Windows NT kernel does not export a well-defined interface to device drivers for modifying the priorities of arbitrary threads, (only for boosting the priority of driver created system threads [10]), we were forced to use an undocumented internal interface for thread priority modification. By using a tool called "NTExport" [11] that uses the symbol information distributed with every build of Windows NT (intended for kernel debugging support) to build an export library for the kernel, we exported the appropriate calls to our driver, enabling thread priority modification. (We hope to find a more portable yet equally effective approach to solve this problem.) When a thread needs to be scheduled, the driver's interrupt handler affects a priority boost for the thread.

In addition, monitoring thread scheduling activity is another key function of the device driver. To provide thread scheduling (current thread) information to the Myrinet Firmware, our device driver exploits a kernel callback on each thread context switch to write the context switch information to the Firmware. Thus, the firmware has precise current thread information.

3. Performance Results

We have implemented DCS for Windows NT 4.0 on a cluster of dual-processor Micron brand Pentium Pro machines running at 200 MHz, each with 64 MB of physical memory. Each machine contains a Myrinet PCI interface connected to a Myrinet switch.

Our experimental methodology was as follows: We ran trials of FM's latency benchmark along with zero, one, two, four, and eight CPU-bound competitor threads, passing one million packets on a round trip between a sender and receiver node. We ran ten runs of this test. Each trial reported a histogram of round-trip times in microseconds. We computed the mean value of each bin for each number of competing threads over the ten trials to get the results we report here.

Preliminary results show that DCS improves performance, but that balancing fairness with performance is a tradeoff. We ran a ping-pong latency microbenchmark and a barrier benchmark on a cluster of six dual-processor 200 MHz Pentium Pro machines. Malfunctions in our LANai development tools prevent us from reporting the results of the barrier series here.

Figure 2 shows the wall-clock time-to-completion for FM's latency test with DCS enabled using E=0 and E=-5 and with DCS disabled. Our testing has indicated that for a large number *n* of compute-bound competitor threads, say four, the NT Server scheduler is fair; that is, we observe each competing thread to receive 1/*n* of the system. For four or more competitors, Figure 2 shows DCS with *E*=-5 to exhibit behavior similar to that of the NT scheduler alone. Since the latency test measures the wall-clock round-trip time required for a series of messages (in this case, one million), the time required for the entire test to run is an indicator of the average round-trip latency observed.

Figure 3 shows the distribution of round-trip times for the eight-competitor case. The graph illustrates the large contributions that round-trip times as large as 1 second make to the latency benchmark's time to completion as the number of competitor threads grows. Most of the contribution that the aggressive DCS configuration (*E*=0) makes to time to completion is clustered on the left side of the graph; non-DCS and the more passive DCS configuration (*E*=-5) show substantial numbers of round trips that last longer than 2 ms to total time to completion. The average latencies for the non-DCS and passive DCS cases swamp the average latency reported in the aggressive DCS case.

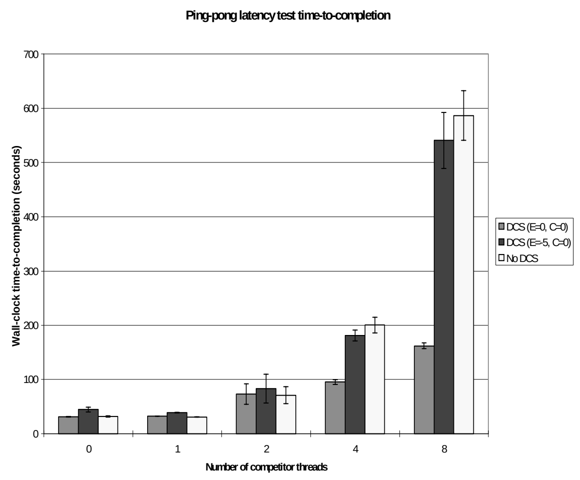

Figure 2. Latency test time-to-completion

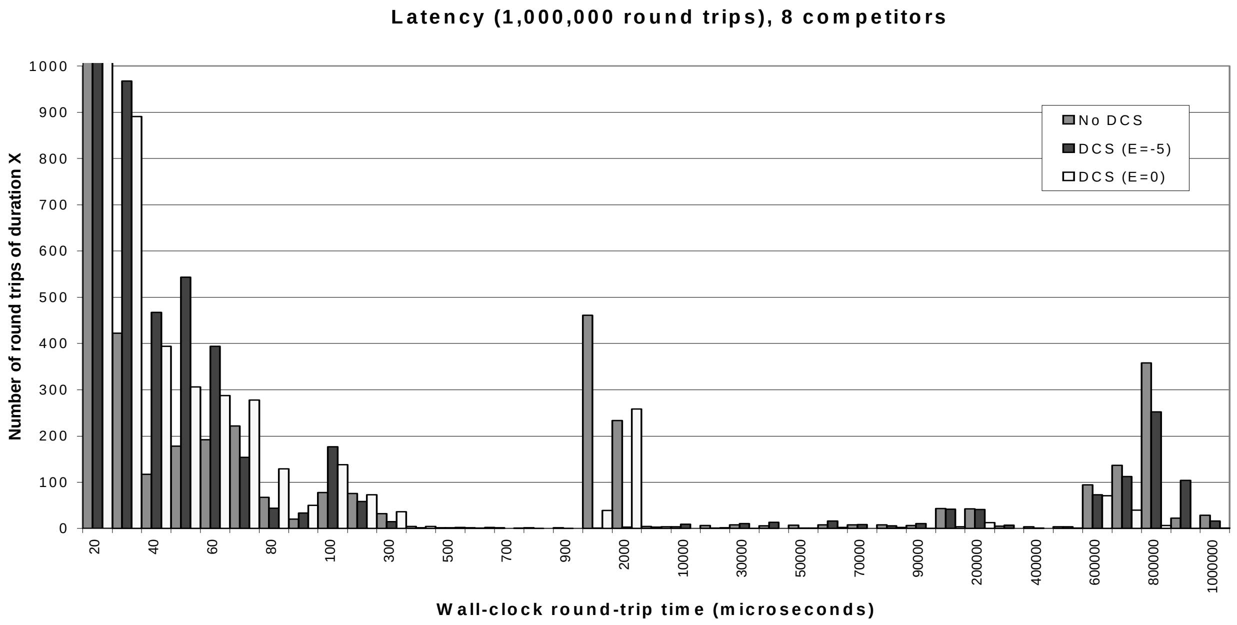

Figure 3. Distribution of latency test round-trip times. For x=20, y=996000 for each configuration.

4. Summary

Our initial performance results are encouraging and suggest that DCS can be implemented and can achieve coscheduling for Windows NT systems. This coscheduling is demonstrated in the improved performance of our benchmark for communicating threads. Unfortunately, we can only report limited results at this point, but hope to report performance data from a broader array of experiments in the near future.

5. Discussion and Future Work

More complete performance measurements using larger applications with our DCS implementation are clearly an important step. Exploration of the parameter space for our DCS fairness equation and techniques for auto-calibration are of interest. In addition, we have added a blocking primitive to the FM interface that we will use to explore the behavior of spin-block synchronization under NT in addition to the current spin-only synchronization, alone and in the presence of DCS. Beyond that, experiments with multiprocessor nodes, proportional share scheduling, and scheduling a broader array of cluster resources are all challenging directions.

Our experience with external customization of the Windows NT scheduler has mixed results. While we initially believed that the wealth of callbacks and external hooks for NT would make external customization easier, our experience was much less encouraging. The callbacks for thread scheduling were inadequate, and only available in the multiprocessor released kernel. For research such as we have discussed to proceed without NT kernel modificaitons, general, better external access to NT's policies (and mechanisms) must be achieved. Priority boosts are a crude *mechanism* for achieving coscheduling, but an effective callback would influence the scheduler's *policy*, possibly achieving the longer-term scheduler synchrony across the cluster that is our goal. A less ambitious approach would involve simply better access to mechanisms for thread priority modification, obviating the need for recourse to tools like NTExport.

More information

More information is available on our WWW site at http://www-csag.cs.uiuc.edu.

Acknowledgments

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grants MIP-92-23732, NASA grant NAG 1-613. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, and Microsoft is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

6. References

[1] Ousterhout, J. K. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22-30, October 1982.

[2] Von Eicken, T, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

[3] Russinovich, M. Differences between Windows NT Workstation and Server. Available from `http://www.ntinternals.com/tune.txt`.

[4] Sobalvarro, P. G. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1997.

[5] Feitelson, D. G. and L. Rudolph. Coscheduling based on run-time identification of activity working sets. In *International Journal of parallel Programming*, Vol. 23, No. 2, pages 135-160, April 1995.

[6] Dusseau, A. C., R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *ACM SIGMETRICS '96 Conference on the Measurement and Modeling of Computer Systems*, 1996.

[7] Sobalvarro, P. G. and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the Parallel Job Scheduling Workshop at IPPS '95*, 1995. Available in Springer-Verlag Lecture Notes in Computer Science, Vol. 949.

[8] Von Eicken, T., A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995..

[9] Tezuka, H. A. Hori, and Y. Ishikawa. Design and implementation of PM: a communication library for workstation clusters. In *JSPP*, 1996.

[10] Microsoft. Windows NT device driver kit documentation.

[11] Russinovich, M. and B. Cogswell. NTExport documentation. Available from `http://www.ntinternals.com`..

[12] Custer, H. Inside Windows NT. Microsoft Press (Redmond, WA), 1993.

[13] Boden, N., *et. al.* Myrinet—a gigabit-per-second local-area network. In *IEEE Micro*, pages 29-36, February 1995.

[14] Pakin, S., Karamcheti, V. and Chien, A. A. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPP's, *IEEE Concurrency* 5(2), April 1997, pages 60-73.

[15] Chien, A., *et. al.* High Performance Virtual Machines (HPVM): Clusters with Supercomputing Performance and API's, Proceedings of the Eighth SIAM Conference on Parallel Processing, March 1997, Minneapolis, Minnesota.