Predictable Network Computing

using

Message-driven Scheduling

Holger Karl, Andreas Polze, Matthias Werner
Department of Computer Science
Humboldt-University of Berlin
10099 Berlin, Germany
{karl, apolze, mwerner}@informatik.hu-berlin.de

Abstract

Clusters of networked, off-the-shelf workstations are currently used for computation-intensive, parallel applications. However, it is hardly possible to predict the timing behaviour of such systems or to give guarantees about execution times. In this paper, we present the design of message-driven scheduling to integrate Quality-of-Service guarantees in the distributed execution of parallel shared-memory programs in the "Shared Objects Net-interconnected Computer (SONiC)" environment.

SONiC provides a class-based programming interface; it consists of user-space servers and a runtime library. SONiC's Scheduling Server allows for predictable partitioning of the CPU cycles of a single workstation. To provide predictable execution in a distributed environment, we extend this approach by a real-time communication server responsible for delivering QoS guarantees to the SONiC tasks by integrating message processing and scheduling.

We present measurements which show the impact of the Scheduling Server on program execution. Furthermore, we evaluate how the communication behaviour over ATM is affected by varying background loads and application priorities on the end systems.

1.0 Introduction

The "Shared Objects Net-interconnected Computer (SONiC)" provides a programming framework for the execution of shared-memory parallel programs within a cluster of off-the-shelf workstations. SONiC uses replicated shared objects as its programming model and provides constructs for distributed synchronization to the programmer of parallel applications. Ease of programming is achieved by encapsulation of communication and synchronization operations within a special class library's implementation. SONiC consists of a number of user-space servers and a runtime library. It has been implemented on the Mach operating system.

In contrast to classical parallel computers, which provide a controlled environment for program execution, a workstation cluster's performance is also affected also by an interactive user's resource needs. Our Scheduling Server - a component of the SONiC system - ensures the requested interactive behaviour of a single workstation while simultaneously giving a specified amount of CPU cycles to parallel tasks. We present measurements showing the Scheduling Server's effect on program execution. However, the issue of resource partitioning in the distributed case is still an open question.

Interactions between the communication sub-system and the local scheduler play a fundamental role in the execution of parallel programs in workstation clusters. To achieve predictable behaviour of distributed parallel applications, guarantees on the communication system's performance are required. Within this paper we describe a design for integration of Quality-of-Service guarantees into the SONiC system. We propose a message-driven approach to scheduling of distributed tasks. We sketch a possible implementation using SONiC's Scheduling Server.

A number of measurements performed in our ATM test environment show the impact of a machine's load and a task's priority on the communication behaviour. In particular, measurements show that communication latency is hardly predictable if a message's receiver is running as a low-priority task on a loaded system. In contrast, increasing the receiver's priority not only lowers the communication latency, but also decreases its standard deviation. Therefore, we expect significant performance improvements through our approach, which closely couples the communication sub-system and the SONiC Scheduling Server.

The remainder of the paper is organized as follows: Section 2.0 describes SONiC's overall architecture. In Section 3.0 we discuss replicated shared objects - SONiC's programming model. The partitioning of a workstation's CPU cycles is described in Section 4.0. In Section 5.0 we develop the concept of a predictable communication sub-system for SONiC. Measurements and implementation issues are covered in Section 6.0, whereas Section 7.0 gives an overview of some related work. The paper concludes with Section 8.0.

2.0 Overall System Architecture

We have developed the "Shared Objects Net-interconnected Computer (SONiC)" as a platform for execution of parallel programs in networked environments. SONiC provides an object-based distributed shared memory system together with synchronization constructs and a remote execution server. The system uses replicated shared objects as its programming paradigm and allows encapsulation of communication and synchronization operations within an object's implementation. It provides functionality for running parallel (multi-threaded) applications within a network of workstations. A scheduling server allows execution of an applications sub-tasks in a pre-determined fashion. SONiC is based on the Mach operating system.

The "Shared Objects Net-interconnected Computer" consists of a programming library - a C++ class hierarchy for replicated objects - and a number of user-space servers that implement the SONiC runtime system as depicted in Figure 1.

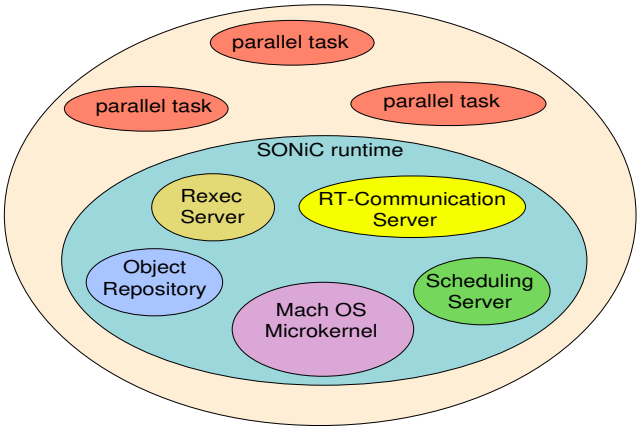

FIGURE 1. Structure of the SONiC runtime on a single node

SONiC treats C++ objects in different address spaces as replicas that form a single shared object. Weak memory consistency protocols are used to maintain a global view on a shared object. SONiC employs *the release consistency* and *entry consistency* protocols to reduce consistency-related communication. The *"Object Repository"* is a user-space server that maintains for every shared object a list of its replicas. Synchronization operations like *acquire_write_lock()*, *acquire_read_lock()*, and *release_lock()* are implemented in a shared object's base class. These operations invoke communication between the holders of an object's replicas and the repository and implement negotiation of access rights.

The *Remote Execution Server (Rexec)* is another component of the SONiC system. It is implemented by cooperating server tasks that run on every node of SONiC's virtual parallel machine. The Rexec-server can be accessed by the library functions *rexec()* and *rwait()* and supports a fork/join-style of parallel programming. Rexec employs a heuristic for load balancing and task placement.

Tasks which are started by the Rexec-server are executed under control of the *Scheduling Server.* The Scheduling Server implements a notion of "contract" between the owner of a workstation (the interactive user sitting in front) and the SONiC system. With such a contract "the owner allows the SONiC system to consume up to *n* percent of the workstation's CPU cycles". However, the Scheduling Server also ensures that a parallel task gets the negotiated amount of CPU cycles, even if the load on the system (the number of "interactive tasks") increases. This way loosely synchronized execution of parallel tasks on different nodes of SONiC's virtual parallel machine can be implemented.

We plan to extend this set of services by a *Real-Time Communication Server (RT-Comm).* This server has the responsibility to provide predictable communication delay and bandwidth between distributed parts of a SONiC application. It relies on Quality-of-Service guarantees given by the network and closely interacts with the Schedul-

ing Server to transfer messages on time from the network interface to the application methods. A suggested design of this server is explained in detail in Section 5.3. This way loosely synchronized execution of parallel tasks on different nodes of SONiC's virtual parallel machine can be implemented.

In the current version of SONiC, the servers communicate using standard Mach IPC (interprocess communication). The access to these server's (e.g., the Object Repository) is completely hidden to the user of SONiC. The functionality to interact with these servers is included in SONiC's base classes that form the part of the runtime system included in the address space of the application. Therefore, the basic design is very portable and only relies on standard communication mechanisms. Some servers might need additional operating system capabilities (like the Scheduling Server, see Section 4.0), but this affects only the enhanced services of this server and doesn't impede the possibility to use SONiC as a whole.

3.0 The Programming Model: Shared Objects

Classical distributed-shared-memory (DSM) systems ([2] present an overview) rely on equally sized memory pages as units of sharing. Often those pages are relatively large chunks of data. Memory pages are not related to the concept of accessing memory through variables as supported by programming languages. Thus, the problem of false sharing may occur: several logically independent variables may reside in the same memory page and are treated like a single, big shared variable.

Weakly consistent data models can dramatically influence the amount of communication needed for execution of a parallel program and for keeping the illusion of a shared data-space alive, a fact which is especially important in distributed environments. However, to make weakly consistent models work, the programmer has to include "hints" to the memory management in his software. Synchronization variables have to be explicitly assigned to (sets of) shared data items. Operations on those variables like *acquire* and *release* have to be placed around accesses to shared data items.

Objects as implementations of abstract data types - which integrate a type with operations and equations on this type - have been established as a programming concept in a number of sequential languages. We believe that objects are the right place to integrate shared data items and the synchronization variables necessary to protect them as well. Our approach provides C++ classes whose instances can be used as simple as ordinary C++ objects in a sequential program. However, we provide a runtime system which treats objects in different address spaces as replicas. Thus, those objects can be seen as shared objects. Weakly consistent memory models are used to update all the replicas that make up a particular shared object.

Within SONiC the programmer has full control over shared objects. He can use C++ language constructs to describe size and layout of objects, the units of sharing. Simply by using different base classes when instantiating an object, the programmer can determine which memory consistency protocol should be used for managing a particular object. No false sharing of data can occur, unnecessary communication is avoided.

Within a DSM-system, replication allows efficient data accesses in case of read sharing. However, this approach raises the cache consistency problem. Programmers often assume that memory is sequentially consistent [3]. But even in sequentially consistent systems, explicit synchronization is required for complex operations. So, most parallel programs define their own higher-level consistency requirements.

These observations have led to a class of weakly consistent protocols. Among them are processor consistency [4], weak consistency [5], release consistency [6] and entry consistency [7]. Such protocols distinguish between data accesses and synchronization accesses. The only accesses that must execute in sequentially consistent order are those relating to synchronization. Updates to shared data in distributed memories can be handled asynchronously.

Within SONiC we support release consistency and entry consistency for shared objects. An association between data and synchronization variables is made on the basis of objects: each shared object is implicitly tied to a synchronization variable.

SONiC class library

Our programming approach provides a whole hierarchy of classes whose instances are replicated shared objects. The base class *SOM* (Shared Objects Memory) implements management functions for shared objects. Within this class, code exists for communication between all the instances - distributed over different address spaces - that make up a shared object in our system. Class *SOM* ties a shared synchronization variable to each data object and provides functions for sending update or invalidation messages to an object's set of replicas. By interaction with the "*object repository*" (a component of the SONiC runtime system) class *SOM* maintains the notion of read- and write-access rights to an object.

The classes *ENTRY_CONS* and *RELEASE_CONS* are derived from class *SOM*. Both classes implement weakly consistent memory protocols and provide the public member functions *acquire_write_lock*, *acquire_read_lock*, and *release_lock*. The programmer who implements a class of shared objects can use these functions to initiate synchronization operations where necessary. By simply choosing the appropriate base class the programmer can decide which consistency protocol his objects should be based on. Figure 2 shows a part of SONiC's class hierarchy.

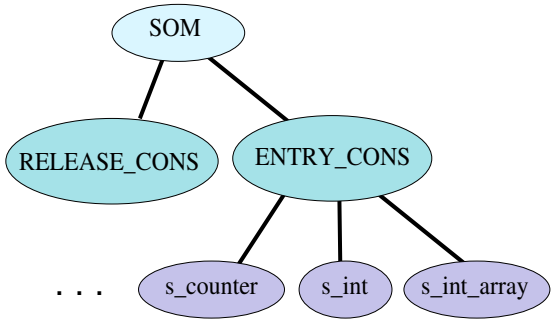

FIGURE 2. Class Hierarchy for Shared Objects

From the application programmer's point of view shared objects appear just like ordinary sequential C++ objects. The programmer only has to watch out for the appropriate public member functions which allow him to access a shared object. Since the implementation of those functions hides all synchronization-specific code, the programmer can treat each shared object as sequentially consistent, although weakly consistent memory protocols are used for its implementation.

4.0 The Scheduling Server - Resource Partitioning

The Mach operating system implements a number of different scheduling policies, including fixed-priority scheduling. Tasks executed under that policy do not experience aging and keep control over the processor until their adjustable quantum (time slice) expires and a higher-priority task becomes ready. Furthermore, Mach implements systems calls that allow manipulation of a task's priority.

Our scheduling server is a user-space task, which runs at the highest priority available in Mach. The server usually sleeps, thus allowing normal operation of the system. SONiC tasks may register as clients with the scheduling server. These tasks are scheduled using the fixed-priority policy. Usually, they execute at a low priority or are even suspended, leaving the rest of the system and the interactive user's tasks undisturbed. From time to time the scheduling server picks one of its client tasks and raises the task's priority to a value which is above the interactive and kernel task's priority. Then, it suspends itself, giving all the CPU cycles to the selected SONiC task (see Figure 3). Later, the scheduling server resets the SONiC task's priority and gives the rest of the system a chance to run again.

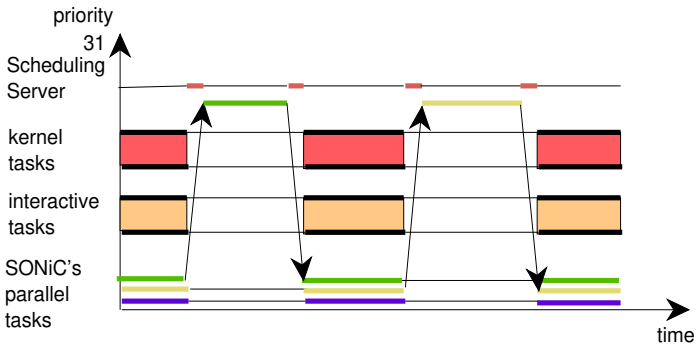

FIGURE 3. SONiC Scheduling Server

Figure 3 demonstrates how priorities are manipulated by the scheduling server. One may notice that kernel and interactive tasks (those not belonging to SONiC) are executed within fixed intervals - independent of the number of parallel tasks. In Figure 3 shaded rectangles represent the interactive and kernel task's phases of activity. The

Scheduling Server currently treats all parallel tasks as having the same priority. An extension of the Scheduling Server will allow the assignment of different priorities to (parallel) client tasks. This way the "*Rate Monotonic Scheduling*" policy can be implemented. *"Earliest Deadline First*" is another real-time scheduling policy that can be implemented based on our Scheduling Server.

Porting the Scheduling Server depends on the underlying operating system's capability to provide real-time scheduling and also on the availability of hand-off scheduling (which can be found in different disguises in a number of systems). Given these properties, porting shouldn't be too difficult. We intend to port the Scheduling Server to Windows NT along with the rest of the SONiC system.

5.0 Predictable Communication

The previous section showed our approach to predictable allocation of processing resources in a single standard workstation. The necessary communication between distributed nodes adds another level of complexity to the problem of predictable execution. This section describes our research goals in connection with predictable communication support for distributed shared memory as implemented by the SONiC system.

5.1 Needs for predictable communication

Predictable communication is important for a number of reasons. First, a problem's semantic may require a priori known and bounded message delay and guaranteed minimal bandwidth. These parameters are subsumed under the term "Quality-of-Service" (QoS). Such requirements are typically found in hard real-time applications like automation or distributed control, but also become more and more prevalent in new areas like distributed multimedia applications (e.g., teleconferencing).

But even performance-oriented distributed computing requires predictable communication. As we assume that distributed programs have to share the computing resources with interactive users, co-scheduling of these distributed tasks should be beneficial to performance. In the "Bulk Synchronous Processing (BSP)" model [29], extra costs are incurred if messages arrive late (cp. Figure 4). This effect is worsened by use of the scheduling server. This is due to the following reason: if a task is in it's high-priority phase, incoming messages destined for a different tasks are not delivered because execution of the consumer of the messages - a user space process - may be suspended by the scheduling server. Thus, any message not delivered before the beginning of a high-priority phase is delayed to the next phase. Additionally, the low-priority phases of SONiC processes offer a natural opportunity to hide the network's latency. Here, the latency immediately ties in with the periodicity of SONiC processes. If messages are delivered late, reserved processing capacity is wasted.

It is important to note that these arguments hold only if the communication is predictable between *applications*, and not only between *hosts*. As we will see below, even networks that provide QoS-support commonly only do so between the respective network interfaces. It is up to the operating system and the runtime system to deliver messages to the application on time. To the best of our knowledge, little work has been done in this area, compared to the large amount of QoS-related work for networks proper [8].

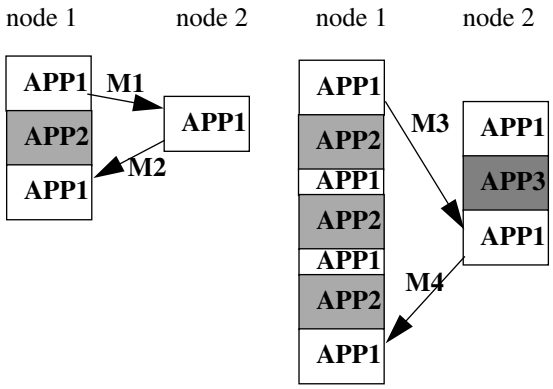

FIGURE 4. Two scenarios: messages M1 and M2 arrive on time; messages M3 and M4 are delayed, thus causing several slots to be wasted by APP1 on node 1.

5.2 ATM - promises and challenges

Traditional real-time communication work often uses dedicated hardware and/or software for achieving guaranteed delivery times. SONiC is intended to work in off-the-shelf networking environments. Today, such environments (like Ethernet, TCP/IP) cannot deliver any Quality-of-Service guarantees. The asynchronous transfer mode

(ATM, [9]) holds the promise of becoming a wide-spread, well standardized, high performance networking environment capable of handling various QoS-requirements.

In an ATM network, an application can request a certain QoS from the network environment and enters a traffic contract with the network. The ATM system guarantees the Quality-of-Service as long as the application fulfills its part of the contract. But this guarantee only applies to the traffic between the local and remote network interfaces! There are no assurances given for the service seen between the two parts of the application itself. So, to achieve virtual synchrony between distributed parts of an application we need to be able to control the timely behaviour of messages not only in the network, but between the application's parts. This is even worse when the application runs under control of the Scheduling Server and is therefore not able to react to incoming messages immediately.

5.3 Integration of predictable communication into SONiC

To integrate a network capable of Quality-of-Service guarantees into the SONiC distributed environment and to provide these QoS properties to the SONiC applications, we envision the following architecture.

The communication protocols are implemented in the SONiC objects themselves, thus enabling user-level communication protocol processing, bypassing the kernel routines wherever possible. This has to be smoothly integrated in the standard access methods to the network interface to ensure that standard applications can still run on the computer unaffectedly. We plan to integrate special methods for packet filtering into our forthcoming ATM-drivers for Mach. An obvious approach is to use the VPI/VCI values[1] to identify the receiving process for an incoming message. Additionally, a protocol header might include fields that specify information to a scheduler on a receiving computer as to when to process these messages and what methods to call to integrate the received data in the computation (similar to [10] as far as method selection is concerned, but extending it by explicit scheduling).

The important point here is to integrate the protocol processing and the Scheduling Server. On a sender side, this gives us the opportunity to charge the application with the processing costs for the communication when run under the Scheduling Service. The receiver side is more complicated. Upon message arrival, a message must be accepted from the network adaptor. The network driver can inform the Scheduling Server which decides to deliver the message directly to the application.

However, it might be the case that the receiver of a message is not eligible for execution, because other applications might be running at this moment. In this case, the messages might have to be queued and delivered later. This has several implications for the concept of virtual synchrony. First, the Scheduling Server must be integrated in the call admission process. It has to check if it is possible, considering guarantees given to other applications, to deliver messages to the application within specified time bounds. Possibly, this results in constraints on when a message must be sent to insure that it can be delivered within a corresponding running phase. Otherwise, only delivery after the execution of other slots can be guaranteed (cp. Figure 5). This has the paradoxical consequence that early message arrival is not desirable, because they either block buffer capacity (as they are not processed immediately) or they may interfer with the execution of other applications.

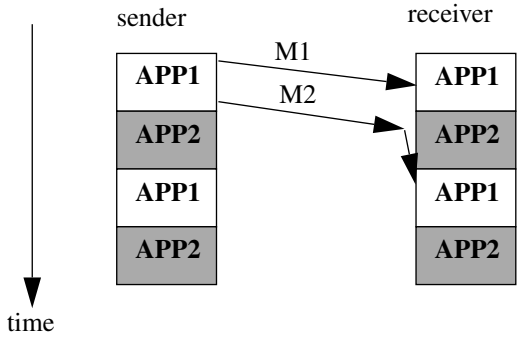

FIGURE 5. Message M1 is delivered immediately, Message M2 is delayed by the execution of App2 on the receiver's side.

An additional advantage of this integration of communication protocols into SONiC is the possibility to directly interact with SONiC's memory management; we will investigate the possibilities of zero-copy protocols, application level framing [11], and other implementation techniques. Also, the integrated protocol processing should, to a

1. VPI=Virtual Path Identifier, VCI=Virtual Channel Identifier. These values together define an ATM-channel.

certain extent, facilitate the adherence to the traffic contract for the application, relieving it from the burden of checking traffic rates, etc. Traffic shaping by the runtime system and copy-efficient protocols are two antagonistic goals, we are looking for an optimal compromise between them.

5.4 Open problems

There are numerous open problems. One major difficulty we foresee is the granularity of the timers used by the Scheduling Server. This will make the implementation of protocols in the user level much more complicated. Another problem is finding a set of experiments that can decide if this concept is valid or if the implementation is sufficiently efficient and correct. Also, the benefits incurred by this method could be rather small, so identifying applications that profit the most is also important.

6.0 Measurements

A task running under control of the scheduling server experiences a high-priority phase ($t_{highprio}$) and a low-priority phase ($t_{lowprio}$). We have investigated the impact of varying values for the scheduling server's period ($p = t_{highprio} + t_{lowprio}$) and for the ratio between the low-priority phase and period ($f = \frac{t_{highprio}}{p}$).
Figure 6 shows the results for an experiment evaluating the performance of the Linpack benchmark when run under control of the Scheduling Server. Linpack evaluates the floating point performance of a CPU by solving a system of linear equations. We have implemented a version of Linpack that counts wall clock time instead of a task's CPU and system time. Therefore, our version of Linpack measures the amount of floating point operations available to a particular task instead of the CPU's performance.

We have run our version of Linpack as standard Mach task (priority 10) and under control of our scheduling server. Figure 6 compares the results obtained from an unloaded system that hosted no running interactive tasks except for the window server, with results from a loaded system, where a graphical animation program was run in the background (with priority 10). The Scheduling Server's period in our experiment was p = 100ms. We have varied the ratio f between low-priority phase and period from 1.0 to 0.1.

Main result of our experiments is, that the number of operations available to a task decreases linearly with the ratio between the low-priority phase and scheduler's period. Also, it shows that the amount of cycles available to the Linpack program is not affected by the "interactive load" on the machine if the Scheduling Server is active. On the other hand, without the scheduling server, performance measured by Linpack drops to one third, even with a relatively small background load.

Additionally, we have investigated how the performance measured by our Linpack program is changed by varying background loads. We have run from 0 to 10 background processes and have observed a standard deviation of 0.365 for the Linpack's performance when run with 40% of the CPU cycles (see Figure 6).

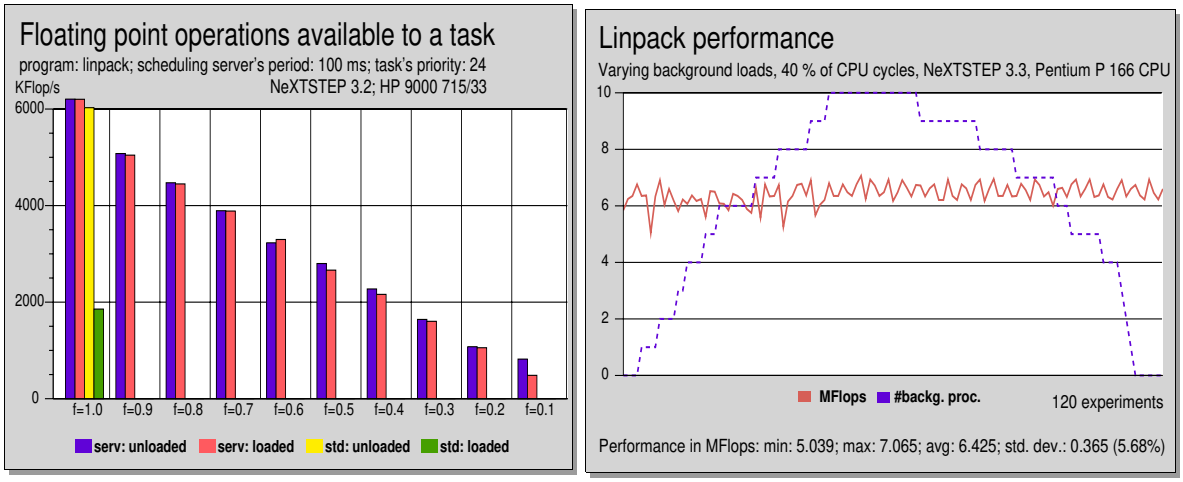

FIGURE 6. Scaling operations available to a task

We have implemented the Scheduling Server under NeXTSTEP 3.3 (Mach 2.5) on HP 715 and Intel Pentium 166 computers. From the interactive user's (subjective) point of view, SONiC tasks seem to be tolerable if they do not consume more than 60-70% of the CPU cycles and if the period of the scheduling server is not higher than p = 500ms.

In a different set of experiments, we evaluated the average round trip time of short packets over an ATM network using varying priority levels of communicating programs and background load respectively. Due to the unavailability of ATM drivers for our hardware, we used two SparcStation 10 running Solaris 2.4/2.5 with Fore ATM-adaptors connected back-to-back. We used a simple client-server program to send packets (12 bytes long) between the system and Linpack again as background load. As priority levels we used standard priority 0 and a nice level of -10 to approximate high-priority tasks. Figure 7 shows the average and standard deviation of the roundtrip time, where either all tasks run at priority 0 or the high-priority programs at nice-level -10.

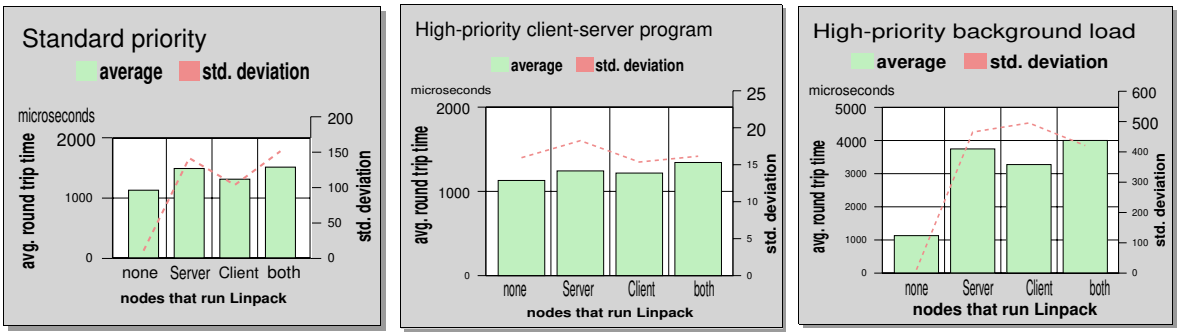

FIGURE 7. Average round-trip time for different priority settings of client-server program and background load.

Obviously, the client-server program is delayed by background load. But the main point is the standard deviation: in the case of high-priority background load, the round-trip time between the distributed parts of an application varies considerably. But as we have seen above, we need high-priority tasks to realize the concept of a Scheduling Server. This means that we cannot give any guarantees for message delivery times for tasks running under the Scheduling Server in low-priority phases. Therefore, this experiment supports our claim that an integration of Scheduling Server and communication system is necessary.

7.0 Related Work

7.1 Distributed execution

A number of projects based on the principle of using a collection of interconnected machines as a concurrent computing platform have been developed.

Many of those systems are based on the message-passing programming model, with PVM [20] as the best known representative. A comprehensive list of such systems can be found in [21]. In contrast to explicit message-passing, the shared-memory programming model provides transparent data communication. A number of distributed shared memory implementations have been described in literature, an overview can be found in [1].

The MUNIN system [22] was one of the first software DSM systems that used release consistency as a model of memory coherence. MUNIN offers multiple consistency protocols. MIDWAY [7] proposes the entry consistency model of page coherence. It tries to minimize communication costs by exploiting the relationship between shared objects and the synchronization variables. However, within MIDWAY entry consistency depends on the correct use of synchronization primitives throughout the whole parallel program. The PANDA system [23] provides a page-based distributed shared memory together with a C++ user-level thread implementation based on a pico-kernel. PANDA employs page differencing to reduce communication bandwidth requirements when transmitting shared memory pages.

A remote execution facility is essential for parallel computing in workstation clusters. A number of systems implement this facility. Among them are the Remote Unix system [24], the Crystal system [25], Condor [26], the Process Server [27] and the DAWGS system [28]. These systems focus on usage of idle workstations for distributed computing purposes. The systems present different approaches to load-placement and load-balancing, such as bidding algorithms, distributed task schedulers, process migration and checkpointing facilities. The techniques used rely on the assumption that a particular workstation is either completely idle or somehow used (interactively). In the latter case, the systems try to release the workstation completely.

In contrast, our scheduling server approach allows partitioning of a single workstation between interactive and parallel tasks. The scheduling server controls the amount of CPU cycles that SONiC's parallel tasks may obtain and provides some notion of "workstation availability" to the interactive user. Using the Scheduling Server

approach we are able to build a virtual parallel computer out of off-the-shelf components that can deliver real-time services.

7.2 Real-time Networking

In real-time research, considerable work has been done on networks capable of guaranteeing delivery times of messages. Typically, this research deals with making access to a shared medium deterministic. Protocols can be based on a priori reservation like TDMA (MARS [12] is classical example), a token bus (IEEE 802.4 [13]) or a deterministic version of Ethernet (802.3D [14]). An overview of this area can be found in [15].

In more recent work, the analysis of response times in switched mediums, like ATM, has attracted more interest. Different scheduling policies for network switches have been proposed ([16] [17] [18] [19]). This work is the justification for assuming that the network can provide Quality-of-Service guarantees, especially bandwidth and bounded delay.

While most of the work relating to Qualiy-of-Service has concentrated on specific network layers, bounding message delay between applications has received comparably little attention. [8] gives an overview of QoS architectures mainly geared towards distributed multimedia applications.

7.3 Coscheduling

Different forms of scheduling across computer boundaries have been introduced. Gang scheduling coordinates the execution of distributed threads of one application (a *gang*) to execute at the same time. [30] shows that gang scheduling is beneficial for fine-grain computation, [31] describes an implementation of gang scheduling on an IBM SP-2. But gang scheduling is rather inflexible, running threads even when they do not want to communicate; additionally, the scheduling granularity in [31] is anticipated to be *several minutes*. The "demand-based" coscheduling approach presented in [33] overcomes this limitation, but uses (in the form suitable for clusters of workstations) preemtion of running processes to achieve coscheduling. This is not compabtible with our demand for guaranteed performance of running jobs and can, in some circumstances, impede the interactive performance of a workstation.

The integration of the two fundamental methods of accessing the network interface, polling and interrupts, with the thread management is investigated in [32]. The authors propose to poll the network whenever the distributed application is idle and to use interrupts for incoming messages otherwise. The decision when to use either method is left to the scheduler. They show that this concept performs comparably well as polling- or interrupt-based methods while releaving the programmer or the compiler of the burden of setting polling rates, etc. Still this method does not meet all our requeirements as we do not want to modify the operating system scheduler itself. Moreover, disabling interrupts while a distributed application is active can be problematic for other applications running on the system.

8.0 Conclusions

We have described the SONiC (Shared Objects Net-interconnected Computer) approach to parallel computing in off-the-shelf workstation clusters. It relies on shared objects as its programming model and provides a fork/join-style of parallelism. In this paper we discussed methods for predictably assigning a specified amount of a cluster's resources to a parallel application.

To this end, we have implemented a Mach-based Scheduling Server. This server determines the amount of CPU cycles that SONiC's tasks will obtain on a particular node. As the Scheduling Server implements some notion of "workstation availability" to an interactive user, SONiC need not be restricted to the use of purely idle workstations. We presented measurements that show the impact of the Scheduling Server on program execution under varying load conditions.

Experiments have shown the influence of high-prioritized tasks on the performance of the communication system and on message delivery times - even using a communication system like ATM capable of providing Quality-of-Service guarantees. Therefore, we have designed our message-driven scheduling approach to integrate the communication subsystem with SONiC's Scheduling Server.

Acknowledgements

Thanks are due to Jan Richling for doing some of the experiments reported on in Section 6.0 and to the anonymous referee for pointing out some work on coscheduling.

References

[1] A.Polze, M.Malek; *Parallel Computing in a World of Workstations*; Proc. of 7th IASTED/ISMM Intl. Conf. on Parallel and Distributed Comp. and Syst., Washington, Oct. 1995, pp. 72-75, IASTED-ACTA Press.

[2] N.Nitzberg, V.Lo; *Distributed Shared Memory: A Survey of Issues and Algorithms*; IEEE Computer, August 1991, pp. 52-60.

[3] L.Lamport; *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*; IEEE Transact. on Comp., C-28(9):241-248, Sept. 1979.

[4] J.R.Goodman, M.K.Vernon, P.J.Woest; *Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors*; Proc. of the 3rd Symp. on Arch. Support for Prog. Lang. and Op. Syst., pp. 64-75, 1989.

[5] M.Dubois, C.Scheurich, F.Briggs; *Memory Access Buffering in Multiprocessors*; Proc. of the 13th Annual Symp. on Comp. Arch., pp. 434-442, 1986.

[6] K.Gharachorloo, D.Lenoski, J.Laudon, P.Gibbons, A.Gupta, J.L.Hennessy; *Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors*; Proc. of 17th Ann. Symp.on Comp.Arch., pp. 15-26, 1990.

[7] B.N.Bershad, M.J.Zekauskas; *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*; Tech. Rep. CMU-CS-91-170, School of CS, CMU, Sept. 1991.

[8] A. Campbell, C. Aurrecoechea, L. Hauw; *A Review of QoS Architectures*; Proc. of the Intl. Workshop on Quality of Service, Paris, March 1996

[9] D. E. McDysan, D. L. Spohn; *ATM : theory and application*; McGraw Hill, New York, NY, 1994

[10] T. von Eicken, V. Avula, A. Basu, V. Buch; *Low-Latency Communication over ATM networks using Active Messages*; Tech. Rep. TR94-1456, Cornell University, Computer Science Department

[11] D. D. Clark, D. L. Tennenhouse; *Architectural Considerations for a New Generation of Protocols*; Proc. of. the ACM SIGCOMM, September 1990

[12] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger; *Distributed Fault-Tolerant Real-Time Systems: The Mars Approach*. IEEE Micro, vol. 9(1), pp. 25-41, 1989.

[13] IEEE 802.4; *Token-Passing Bus Access Method;* IEEE New York, 1985

[14] G. LeLann; *The 803.2D Protocol: A variation of the IEEE 802.3 Standard for Real-time LANs*; INRIA, Technical Report, France, 1987

[15] J. F. Kurose, M. Schwartz, Y. Yemini; *Multiple-Access Protocols and Time-Constrained Communication*; Computing Surveys, March 1984, pp. 43-70

[16] A.K. Parekh, R.G. Gallager; *A generalized processor sharing approach to flow control in integrated services networks: The single node case*; IEEE/ACM Transactioins on Networking, 1(3):344-357, June 1993

[17] A.K. Parekh, R.G. Gallager; *A generalized processor sharing approach to flow control in integrated services networks: The multiple node case*; IEEE/ACM Transactioins on Networking, 1(2):137-150, April 1994

[18] A. Raha, N. Malcom, W. Zhao; *Hard real-time communications with weighted round robin service in ATM local area networks*; Engineering of Complex Computer Systems, vol. 1, pp. 96-103. IEEE, IEEE Computer Society Press, Nov. 1995

[19] S. Jamaloddin Golestani; *A stop-and-go queuing framework for congestion management*; ACM SIGCOMM '90, pp. 8-19, Sep. 1990

[20] V.S.Sunderam; *PVM: A Framework for Parallel Distributed Computing*; J. Concurrency: Practice and Experience, 2(4), pp. 315-339, Dec. 1990.

[21] O.A.McBryan; *An Overview of Message Passing Environments*; Parallel Computing, Vol. 20, No. 4, pp. 417-444, April 1994.

[22] J.B.Carter, J.K.Bennet, W.Zwaenepoel; *Implementation and Performance of Munin*; Proc. of the 13th ACM Symp. on Op. Syst. Principles, pp. 152-164, 1991.

[23] H.Assenmacher, T.Breitbach, P.Buhler, V.Hübsch, R.Schwarz; *PANDA -- Supporting Distributed Programming in C++*; Proc. of ECOOP'93, LNCS vol. 707, 1993, pp. 361-383.

[24] M.J.Litzkow; *Remote Unix: Turning idle workstations into cycle servers*; Proc. 1987 Summer USENIX Conference, Phoenix, AZ, pp. 381-384.

[25] D.J.DeWitt, R.Finkel, M.Soloman; *The CRYSTAL multicomputer system*; IEEE Transactions on Software Engineering, SE-18, 8, pp. 953-966, 1987.

[26] M.J.Litzkow, M.Livny, M.W.Mutka; *Condor -- A hunter of idle workstations*; Proc. of 8th Intl. Conf. on Dist. Comp. Syst. 1988, pp. 104-111.

[27] R.Hagmann; *Process server: Sharing processing power in a workstation environment*; Proc. of 6th Intl. Conf. on Dist. Comp. Syst. 1986, pp. 260-267.

[28] H.Clark, B.McMillin; *DAWGS -- A Distributed Compute Server Utilizing Idle Workstations*; J. of Parallel and Dist. Comp., 14, pp. 175-186, 1992.

[29] L. Valiant; *A Bridging Model for Parallel Computation*; Communications of the ATM, vol. 33(8), pp. 103-111, Aug. 1990

[30] D. Feitelson, L. Rudolph; *Gang Scheduling Performance Benfits for Fine-Grain Synchronization*; Journal of Parallel and distributed Computing, vol. 16, pp.. 306-318, 1992

[31] H. Franke, P. Pattnaik, L. Rudolph; *Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems*; Proc. of the 6th Symp. on the Frontiers of Massively Parallel Computation, Annapolis, MD, Oct. 1996

[32] K. Langendoen, J. Romein, R. Bhoedjang, H. Bal; *Integrating Polling, Interrupts, and Thread Management*; Proc. of the 6th Symp. on the Frontiers of Massively Parallel Computation, Annapolis, MD, Oct. 1996

[33] P. Sobalvarro, W. Weihl; Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors; Job Scheduling Strategies for Parallel Processing (Workshop at IPPS 95), Santa Barbara, Springer, LNCS vol 949, 1995