Predictable Network Computing

Andreas Polze, Matthias Werner, Gerhard Fohler
Humboldt University of Berlin
Unter den Linden 6, 10099 Berlin, Germany

Abstract

Clusters of networked commercial, off-the-shelf (COTS) workstations are presently used for computation-intensive tasks that were typically assigned to parallel computers in the past. However, it is hardly possible to predict the timing behavior of such systems or to give guarantees about execution times. In this paper we show how our SONiC (Shared Objects Net-interconnected Computer) system can control timing and partitioning of a workstation as a step towards a distributed real-time system built from COTS components.

SONiC provides a class-based programming interface for creation of replicated shared objects of arbitrary, user-defined sizes. Weak consistency protocols are employed to improve system's performance.

Our Scheduling Service ensures the requested interactive behavior of a workstation while simultaneously giving a specified number of CPU cycles to parallel tasks. Using off-line scheduling methods we are able to implement real-time Guaranteed Services on COTS workstations.

1 Introduction

Clusters of networked commercial, off-the-shelf (COTS) workstations allow users to benefit from high-performance parallel computing, increased accessible resources, and scalability. A number of practical problems, however, jeopardize the benefits. Cumbersome methods to program parallel applications limit the effective parallelism. Uncoordinated resource management and unpredictable loads impede the performance gains: Workstations will be loaded above tolerable levels to, e.g., the interactive user. No predictions or guarantees about a program's completion can be given. The execution of critical applications cannot be guaranteed off-line. In this paper, we present methods to overcome these practical problems.

We assume the following scenario and requirements: A set of COTS workstations is connected via a network to form a cluster. Each workstation serves as interactive device to a user as well as a computing node in our virtual parallel computer, the "Shared Objects Net-interconnected Computer (SONIC)" [1][2].

Applications are distributed to benefit from high-performance, parallel computing. Without restrictions, workstations may be assigned large numbers of these computations and consequently lack capacity for local programs. Thus, an interactive workstation has to be ensured a minimum amount of resources so response times to interactive user requests are acceptable. We accomplish this by providing a scheduling server, which allows a minimum CPU usage to be specified and kept for a program.

The programming model for the cluster has to support the parallel computing capabilities, yet remain simple and close to common sequential methods to ensure acceptance and use by programmers. We propose the use of shared objects, which include communication and synchronization within the object definition.

A cluster will typically run a set of critical, shared applications, such as a distributed database, programs to read and update news and information services, as well as network and security activities. Postponed executions or delays may cause substantial loss or damage. Therefore, the set of critical applications has to be guaranteed to execute feasibly before the system is deployed. We present a kernel mechanism to provide guaranteed amounts of time and resources for limited types of programs. Part of these slots is reserved for critical applications, which we preschedule with a static real-time scheduling algorithm.

The decision when to start certain programs during the cluster's runtime is dependent on when they will complete. Data consistency checks, version updates of software, or balancing, will only be started when their estimated completion time is acceptable and, once started, can be guaranteed to be kept, independent of, e.g., changing loads later on. We achieve this by designating part of the guarantee slots for such programs. If


* currently at the Software Engineering Institute, CMU, apolze@sei.cmu.edu
† {mwerner, fohler}@informatik.hu-berlin.de
the program’s execution time is available, we use a real-time guarantee algorithm to determine if the program can be completed within a given time. If so, we reserve the resources needed for it in the future guarantee slots.

By limiting the amount of freely available resources and providing guarantee mechanisms for the reserved resources, we can provide a number of assertions to improve the networked workstations’ applicability.

The rest of the paper is organized as follows: Section 2 describes the overall architecture of our system and underlying assumptions. Section 3 focuses on Shared Objects - our programming model. Section 4 discusses our approach for Partitioning of a workstation’s resources (CPU cycles) among a number of tasks, whereas Section 5 discusses the issue of Guaranteed Services. Sections 6 and 7 describe implementation details and present some measurements. Section 8 describes related work. The paper concludes with Section 9, which also presents some ideas for future work.

2 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 service. The system uses replicated shared objects as 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 service allows execution of an applications sub-tasks in a pre-determined fashion. SONiC is based on the Mach operating system.

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

We assume a reliable, time-bounded (i.e., synchronous) communication between the different workstations. That includes a limited times for net-communication and delivering of received messages to the application [3].

The “Shared Objects Net-interconnected Computer” consists of a programming library - a C++ class hierarchy for replicated objects - and a number of userspace server which implement the SONiC runtime system as depicted in Figure 1.

SONiC treats C++ objects in different address spaces as replicas which form a single shared object. Weak memory consistency protocols are used to maintain a global view on a shared object. SONiC employs release consistency and the entry consistency protocol to reduce consistency-related communication. The “Object Repository” is a user-space server which 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 Service (Rexec) is another component of the SONiC system. It is implemented by co-operating server tasks which run on every node of SONiC’s virtual parallel machine. The Rexec-service can be accessed by library functions rexec() and rwait() and supports a fork/join-style of parallel programming. Rexec employs a heuristic for load balancing and task placement.

Figure 2. The graphical user interface to the SONiC System

Tasks which are started by the Rexec-service are executed under control of the Scheduling Server (Sched). The Scheduling Server implements a notion of “contract” between the owner of a workstation (the interactive user in front) and the SONiC system. With such a contract “the owner allows the SONiC system to consume up to x percent of the workstation’s CPU cycles”. However, the Sched-service 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.

In order to make parallel programming in workstation environments easy, one has not only to care for the application programmer, but also for the user of parallel applications. To simplify management of the SONiC runtime system, we have built a graphical front end. The front end allows for easy configuration of the virtual parallel machine within a network of workstations. Furthermore, it supports the startup of parallel applications. Output of all the parallel/distributed threads belonging to an application is displayed in a multiplexed fashion. Figure 2 presents a screen dump from one run of our system. It shows the execution of a parallel Mandelbrot computation.

3 Programming Model: Shared Objects

3.1 Shared Memory versus Shared Objects

Classical distributed-shared-memory (DSM) systems [4] 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 dataspace 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 implementation 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 which 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.

3.2 Consistency Models

Maintaining a predictable view of memory across machines is called memory consistency or memory coherency. If a memory management system implements full consistency, data modified by one processor will be immediately visible to all other processors sharing the memory. Alternately, memory management may implement weaker consistency, in which updates are deferred until absolutely needed or until triggered by a special mechanism.

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 [5]. 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 [6], weak consistency [7], release consistency [8] and entry consistency [9]. 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.

3.3 Class Hierarchy

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. This class implements communication between all the instances - distributed over different address spaces...
- which 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 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 3 shows a part of SONiC’s class hierarchy.

Figure 3. 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 The scheduling server - resource partitioning

The Mach operating system implements a number of different scheduling policies, among them 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 which 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 4). Later, the scheduling server resets the SONiC-task’s priority and gives the rest of the system a chance to run again. Figure 4 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 4 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 typical real-time scheduling policies such as “Rate Monotonic Scheduling” (RMS) and “Earliest Deadline First” (EDF) can be implemented.

Figure 4. SONiC scheduling server

Figure 4 hides a priority-inversion problem which is present in a number of Mach versions. The thread_priority()-system call which is executed by the Scheduling Server to decrease a client task’s priority after a high-priority phase is executed by kernel code at a priority which is lower than 31. Thus, during execution of this system call it may happen, that the client task becomes the one with the system-wide highest priority. Once this happens, the Scheduling Server has no chance to decrease the client task’s priority.

We have developed two approaches to solve this priority inversion problem. The first solution consists of choosing a high-phase priority for the client tasks, which is below the priority range kernel tasks are executed in, but still higher than the interactive task’s priorities. Experiments have shown that priority 24 is a good value on NeXTSTEP systems. The second solu-
tion to the priority inversion problem does not use *thread_priority()-system* call at all. This solution requires an extension to the Mach kernel. Using a “Loadable Kernel Server (LKS)” we have implemented this approach on NeXTSTEP as well. Both implementations are described in a subsequent section. For the experiments presented here we have used the first approach.

5 Guaranteed services

While high performance parallel computing provides for fast execution of applications in clusters of workstations, the demand of common types of programs cannot be met: Critical activities, such as database updates, reading and preparation of input values from sensors or newsfeeds, backup procedures, or security checks, need to execute to completion at fixed times within certain time limits. Other programs, such as data consistency checks, upgrades, or transactions require predictions or guarantees about their completion time to decide whether their start is acceptable at the time under consideration. We solve these problems by controlling the amount of resources given to various types of applications.

**Guarantee slots.** Using the Scheduling Server approach described in Section 4, we are able to provide *guarantee slots*. These are scheduled to run at fixed periods and provide a guaranteed amount of time and resources, accessible only to limited types of programs. Thus, we can calculate the amount of CPU time and resources available for guarantees ahead of time.

Let “*slotlength*” s be the duration of a guarantee slot and “*p*” its period. Then the CPU is reserved for the time intervals \([i\times p, i\times p + s], i = 0,1,2,...\). The amount of reserved time in a time interval \([ts, te]\) has a lower bound of

\[
\left\lfloor \frac{te - ts}{p} \right\rfloor \times slotlength
\]

Given this knowledge, we can apply real-time scheduling methods to manage the programs inside the guarantee slots. From the point of view of the guaranteed programs, we can view the series of guarantee slots as CPU of reduced availability, \(slotlength/p\).

At time \(i\times p\), the operating system scheduler hands over control to the guarantee scheduler and resumes it at \(i\times p + slotlength\) again. In order to allow different guarantee services, we divide the guarantee slots even further.

**Pre-runtime guarantees.** A portion of each guarantee slot is reserved for the execution of critical programs, that need to execute at pre specified points in time, possibly periodically, and require guarantees about their timely completion before the system is deployed. We assume these applications to be scheduled by an offline scheduler [10]. It produces scheduling tables which list the exact start and end of executions. Inside the slot, a simple dispatcher interprets these by starting and suspending programs.

**Runtime guarantees.** The remaining part of each guarantee slot is devoted to executing applications which require predictions of their completion times or guarantees about their timeliness, if started at a point in time unknown beforehand. If these are not satisfactory, the execution may be delayed or abandoned. Again, we make use of real-time scheduling work. Runtime guarantee algorithms like the ones presented in [12] or [11] allow to specify dependencies of programs, resources, and execution orders. Given these and the available resources in the guarantee slots on the workstations in the cluster, the algorithm determines, whether a completion will be within a given time, or can be used to calculate completion times. It provides rules for selecting programs to run inside a guarantee slot and can also direct migrations.

**Prerequisites.** Our methods need operating system provisions for reservation of guarantee slots, reliable time-bounded communication and synchronization of slots between workstations. Furthermore, we need to have knowledge about the guaranteed programs’ properties. In particular, the execution time has to be bounded and known. If tasks access critical sections, synchronize with each other, these have to be known in advance. Critical applications with offline guarantees require knowledge about execution times, patterns, or periods.

**Future.** As a next step, we plan to include statistical predictions into our guarantees to overcome the limitations of advance knowledge. Instead of giving firm guarantees, it may be sufficient to have some probability for its timely completion, given a probability for the execution time is known. Since we provide these within guarantee slots, the probabilities will be independent of the load generated by non guaranteed applications.

6 Implementation issues

6.1 A user-space server

Our first implementation of the Scheduling Server is based upon the concept of a multi-threaded user-space server. It uses two different threads. The first thread handles requests from client tasks (i.e., threads), which can register with the scheduling server by calling a library function `start_scheduler()`. This function transfers the client’s thread control port to the server and inserts it in a list. The function `stop_scheduler()`
de-registers a client thread by removing its thread control port from the Scheduling Server’s list.

Figure 5 shows the essential parts of the second thread - the main loop. Within that loop a client thread is picked from a list (via next_thread()). Then, execution of this thread is resumed (via thread_resume()) and control is handed over to that particular thread (via thread_switch()). Client threads manipulated by the Scheduling Server are executed following Mach’s fixed-priority scheduling policy. Therefore, a client thread runs uninterrupted until its quantum (time slice) expires. Afterwards the Scheduling Server gets control back. Then, it suspends the client thread (via thread_suspend()) and gives the rest of the system a chance to run (via thread_switch()).

```c
while(1) {
    th_id = next_thread(th_list, th_id);
    if (th_id == THREAD_NULL) {
        mutex_lock(th_list_lock);
        while (th_id = next_thread(th_list, th_id)) == THREAD_NULL) {
            condition_wait(new_th_reg, th_list_lock);
        }
        mutex_unlock(th_list_lock);
    }
    if (thread_resume(th_id) != KERN_SUCCESS) {
        th_id = THREAD_NULL; continue;
    } /* handoff scheduling, high-priority thread runs until its quantum expires */
    if (thread_suspend(th_id) != KERN_SUCCESS) {
        th_id = THREAD_NULL; continue;
    } /* give rest of the system a chance, invoke Mach scheduler */
    thread_switch(THREAD_NULL, SWITCH_OPTION_WAIT, time_os);
}
```

Figure 5. Main loop of the scheduling server

In Figure 5 the variable th_list denotes the data structure which contains a list of client thread’s control ports. Every entry in that list also contains the high-phase priority associated with a client and the duration of high- and low-phase. The function next_thread() retrieves a thread from th_list. Calls to next_thread() can be executed without accessing a synchronization variable (mutex). In case of an empty list, the scheduler main loop waits on a condition variable for arrival of new client threads.

In our current version of the Scheduling Server client threads are stopped during their low-priority phase. Therefore, we ensure that regardless of the “interactive load” on a machine a particular client thread gets exactly the negotiated amount of CPU cycles. This way we can “scale down” the computing power of a particular workstation which allows us to “loosely synchronize” SONiC’s compute nodes.

### 6.2 Using a Loadable Kernel Server

As mentioned, the user-space server solution works fine on the Mach 2.5-based NeXTSTEP system as long as client threads ask for a priority less than 24. With a priority of 24 or higher, the client thread ceases to return the control after its quantum elapsed. The reason is probably an anomaly of the NeXTSTEP operating system. Threads with a priority higher than 23 are assumed to be threads belonging to the operating system. These system threads seem to rely on cooperative scheduling. Therefore, we used a Loadable Kernel Server (LKS) that may access Mach kernel’s callout functions at the highest, non-maskable callout level (callout priority 5). Loadable kernel servers are a special feature of NeXTSTEP. The system allows to load kernel space servers and drivers at random time after systems initialization. Our scheduling server consists of three parts: the kernel server loader, the kernel server, and the user library.

![Architecture of kernel scheduling server](image)

**Figure 6. Architecture of kernel scheduling server**

Figure 6 shows a simplified scheme of the scheduling server. The scheduling server loader loads the server, contacts the LKS loader tasks and provides the relocatable server module. The actual kernel server consists of two threads and one callout service routine. The first thread has the same function as the equivalent thread in the user space server. It handles requests from client tasks and from the scheduling server loader, keeps track of the thread list, and starts and stops the callout thread. The callout schedule thread has the highest possible priority (31), so there is a guarantee it will come up in every scheduling period. If it is running, it schedules all calls of the callout service routine during the next scheduling period. After that, it blocks...
itself for the duration of one scheduling period and in that way it returns the control to the Mach scheduler.

In this way, the callout service routine is guaranteed to be called by the Mach system regardless of the running threads’ priorities. The server routine has access to the scheduling server memory map. The task of the callout service routine is to manipulate the priority of (user) real-time threads. To circumvent deadlocks, it does not use the `thread_priority()` function but calculates the address of the thread's `thread_struct` and manipulates the priorities directly. Using our scheduling server implementation based on NeXTSTEP’s LKS concept, we can guarantee the amount of CPU cycles available to a particular task. Even Mach kernel threads cannot prevent real-time threads from running. Therefore, our scheduling server provides the bases for the Guaranteed Services concept presented in Section 5. We are preparing more measurements to evaluate our concepts thoroughly.

7 Measurements

A task running under control of the scheduling server experiences a high-priority phase \((t_{\text{highprio}})\) and a low-priority phase \((t_{\text{lowprio}})\). We have investigated the impact of varying values for the scheduling server’s period \((p = t_{\text{highprio}} + t_{\text{lowprio}})\) and for the ratio between high-priority phase and period \((f = t_{\text{highprio}} / p)\).

![Execution under Scheduling Server](image)

Figure 7. Impact of scheduling server on a program’s runtime

Figure 7 shows runtimes for a task according to different values for \(p\) and \(f\). The overall runtime of a task is inversely correlated to its CPU usage. Therefore, in Figure 7 higher values for the task’s runtime indicate a lower CPU usage. The single task measured in Figure 7 simulates all the parallel tasks generated by SONiC on a particular workstation. Another experiment run under control of the Scheduling Server concerns the Linpack benchmark. 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, not 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 8 compares the results obtained from an unloaded system, which 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 \(p = 100\)ms. We have varied the ratio \(f\) between high-priority phase and period from 1.0 to 0.1.

![Floating point operations available to a task](image)

Figure 8. Scaling operations available to a task

Main result of our experiments is, that the number of operations available to a task decreases linearly with the ratio between 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.

We have implemented the scheduling server under NeXTSTEP 3.3 (Mach 2.5) on HP 715 and Intel Pentium 133 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 CPU cycles and if the period of the scheduling server is not higher than \(p = 500\)ms.

8 Related work

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-pass-
ing programming model, with PVM as the best known representative. A comprehensive list of such systems can be found in [13]. 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, [1] presents an overview.

The MUNIN system [14] was one of the first software DSM systems which used release consistency as a model of memory coherence. MUNIN offers multiple consistency protocols. MIDWAY [9] 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.

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 [15], Condor [16], the Process Server [17] and the DWAGS system [18]. All those 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 which SONiC’s parallel tasks may obtain and provides some notion of “workstation’s availability” to the interactive user. Using the scheduling server approach we are able to build a virtual parallel computer out of COTS components which can deliver real-time services.

The idea of a real-time server has been addressed in [19], for a proprietary architecture and operating system. In contrast to [19] our approach focuses on extension of a COTS operating system without programming the operating systems kernel as demonstrated in RT-Mach [20].

Examples for algorithms which schedule off-line have been presented, e.g., in [21] and [10]. On-line scheduling of tasks with synchronization and resource constraints, without consideration of off-line guarantees, was developed for the SPRING system, e.g., [12].

The use of resources unused by the static periodic tasks has been studied in the context of EDF scheduling, [22], [23] gives online acceptance tests of O(N).

9 Conclusions

We have developed the “Shared Objects Net-inter-connected Computer (SONiC)”. SONiC supports execution of parallel programs on clusters of workstations. It relies on shared objects as programming model and provides a fork/join-style of parallelism. Within this paper we have focused on methods for resource and load management present in the SONiC system.

As a component of SONiC we have implemented a Mach-based scheduling server, which allows for partitioning of a workstation among parallel and interactive tasks. The scheduling server determines the amount of CPU cycles which SONiC’s parallel tasks will obtain on a particular node of a workstation network. Therefore, the scheduling server implements the concept of guaranteed (real-time) services on a COTS operating system. It also implements some notion of “workstation availability” to the interactive user. In contrast to other “remote execution systems” SONiC does not have to be restricted to the use of only purely idle workstations.

Other components of SONiC implement an object-based distributed shared memory and a remote execution service. A class library implements weakly consistent schemes of memory management - lowering the amount of consistency related communication in the system. Consistency and synchronization-related code can be encapsulated in the implementation of a shared class. Therefore, SONiC’s shared objects provide ease-of-use to the application programmer.

We have implemented the SONiC system and our scheduling server concept on the NeXTSTEP 3.3 (Mach 2.5) system on HP 715 and Intel Pentium-based computers. The scheduling server relies on Mach’s fixed-priority scheduling policy. By manipulating tasks’ priorities it overrides the operating system scheduler. We plan to port SONiC and the scheduling server concept to Windows NT as well.

10 Acknowledgments

We thank our student Jan Richling for the help with the measurements. Also we thank Miroslaw Malek for the valuable discussion and the reviewers of this paper for the comments.

11 References

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