Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time Scheduler

Hongyi Zhou

Bellcore
RRC 4C-306, 444 Hoes Lane
Piscataway, NJ 08855-1300

Karsten Schwan Ian F. Akyildiz

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

In this paper, we investigate two questions regarding real-time multiprocessor scheduling for large-scale NUMA architectures: (1) how are the latency and the quality of scheduling affected by different degrees of completeness in the information shared among multiple, potentially concurrent schedulers? and (2) how can scheduling information be represented so that it is efficiently and concurrently accessible? We present a real-time scheduling algorithm for multiprocessors that is scalable in the number of tasks performing scheduling and in the maximum amount of computation time consumed by those tasks. We also develop a flexible representation for shared information within the distributed scheduler that is easily varied regarding its degree of information completeness. We then show that the sharing of incomplete (vs. complete) information can lead to increased performance regarding scheduling latency with few or no losses in scheduling quality. In addition, we show that this holds for a variety of parallel machines, ranging from NUMA to distributed memory machines.

1 Introduction

Recent research in multiprocessor scheduling has focussed on the effects of multiprogramming on system throughput and on the speedup of parallel program execution [1, 12]. Experimental results have been attained for parallel program scheduling on dedicated multiprocessors, such as the work on co-scheduling [6], on real-time multiprocessor scheduling [4, 13, 3, 2], and others. Research in multiprocessor operating systems has complemented such work by designing interfaces among hierarchically structured schedulers and task dispatchers in experimental and Unix-compatible multiprocessor operating systems [1].

In such research, it is often assumed that schedulers – like other operating system components – are internally concurrent so that they can be easily scaled to different size parallel machines and to varying application demands. For large-scale parallel machines, this implies that scheduling decisions are made by multiple potentially concurrent and cooperating tasks. Furthermore, since large-scale machines typically exhibit non-uniform memory access (NUMA) characteristics (or do not offer shared memory at all, as with distributed memory machines), such tasks should access scheduling information locally whenever possible. In other words, scheduling information must be distributed across multiple memory units.

Distribution of scheduling information enables access locality and concurrency during scheduling. However, additional scheduling overhead may result from accesses to remote scheduling information required in several situations, including when a processor's scheduler attempts to schedule a task on a remote processor. Such overhead can be reduced by a scheduler's use of *incomplete information* about other processors' schedules, but usage of incomplete information also results in tradeoffs regarding the quality vs. latency of scheduling decisions. For example, while some scheduling algorithms simply use recent estimates of total workload on remote processors (an example of incomplete information) [9] when performing task to processor assignment, it has been shown that more complete information regarding remote processors' schedules is required if a task must be co-scheduled with other tasks [6].

In this paper, we investigate two open questions regarding multiprocessor scheduling for NUMA architectures:

- *Incomplete scheduling information* – how are the latency and the quality of scheduling affected by different degrees of completeness in the information shared among multiple, potentially concurrent schedulers?

1052-8725/92 $3.00 © 1992 IEEE

- *Efficient information representation* – how can scheduling information be represented on NUMA and distributed memory machines so that it is efficiently and concurrently accessible?

We will address the issues listed above for a certain class of parallel programs: real-time applications executing on dedicated NUMA multiprocessors. Specifically, the parallel applications this research addresses are the dynamic generalizations of real-time applications studied earlier in [4, 11]. In dynamic real-time applications, it is possible to create on-line time-constrained tasks that cannot be predicted or accounted for prior to program execution. Since such *dynamic* tasks' timing requirements are not known prior to program execution, schedulability analysis must be performed on-line. Off-line scheduling algorithms are also required, because in actual real-time systems dynamically created tasks typically co-exist with a minimal statically defined task set. In this paper, we are concerned with on-line multiprocessor scheduling.

Additional contributions of this work include:

- a novel real-time scheduling algorithm for multiprocessors that is designed to be *scalable* in the number of tasks performing scheduling and in the maximum amount of computation time consumed by those tasks; and
- a representation for shared scheduling information – called *load intervals* – that can be easily varied regarding its degree of information completeness.

This paper is organized as follows. In Section 2, we define the scheduling problem being addressed by this work. Section 3 describes the scheduler and its scheduling algorithm. The performance evaluation of the algorithm in Section 4 is carried out on a 32-node GP1000 BBN Butterfly and on a SUN Sparcstation using artificial workloads for the distributed scheduler. Projections of those results to distributed memory machines (like the Intel iPSC series machines) are presented in Section 4.1.3. Section 5 concludes the paper.

2 Problem Description

We are concerned with the problem of allocating and scheduling a set of n independent preemptable sporadic tasks on a shared-memory multiprocessor system with p identical processors. Each processor in such systems has local memory that can be shared with other processors at some penalty in memory access time via a switching network. The time required for the completion of a memory reference therefore depends on the relative locations of the requesting processor and the target memory unit. For the shared memory implementation of the distributed scheduler, we also assume that locally cached data values are subject to hardware-supported shared memory consistency constraints. The BBN Butterfly multiprocessor and hierarchical cache architectures are examples of such systems.

Each task is described by (A, S, C, D), where A is its arrival time, S is the earliest possible time at which its execution may begin (start time), C is the estimated maximum computation time, and D is the deadline by which it must complete its execution, which are all assumed known when the task arrives at the system (or at latest, when it is scheduled). We shall denote the i-th task by T_i and use the subscript i in all of its parameters. The *laxity* l of task T_i on a processor is given by $l = \Delta - C_i$ where Δ is the total available processor time in the scheduling interval $[S_i, D_i]$. Note that since Δ varies with time, a task's laxity changes over time. A task's laxity may be used as a measure of its urgency at some given point in time. For example, a task's laxity of 0 at time t implies that the task must be scheduled immediately in order to meet its deadline. A task's *maximum laxity value* is the maximal value among its set of laxities on the parallel machine's processors.

The function of a scheduling algorithm is schedulability analysis and schedule construction. *Schedulability analysis* determines whether a feasible schedule exists for a set of tasks, whereas *schedule construction* calculates a feasible schedule, if it exists. Schedulability analysis is particularly important in dynamic real-time applications because it determines at run-time whether the timing constraints of newly created tasks can be met. Specifically, we call a newly arriving task *dynamically schedulable* if it can be scheduled to meet its timing constraints such that all previously scheduled tasks also remain schedulable.

Some of the existing real-time multiprocessor scheduling algorithms [4, 13] define a schedule S as a sequence of slices, where each *slice* is a vector of length p (the number of processors). The ith element of this vector indicates the task assigned to run on processor P_i during the time interval defined by the slice's start time and duration (or length). Therefore, a slice describes some subset of tasks that can run in parallel during some time interval.

Given the schedule description above, a multiprocessor scheduling algorithm operates by extending an empty or partial schedule with new slices, one slice at a time [13]. Efficiently accessible global data structures for the slice-based schedule description have been designed for small-scale UMA machines [10]. For large-scale parallel machines and NUMA machines, the important implementation issues that should be addressed include:

- *Concurrency of access* – since multiprocessor

scheduling consists of both (1) schedulability analysis (also called 'scheduling') and (2) task scheduling (also called 'low-level scheduling' or 'dispatching' [12]), tasks performing steps (1) or (2) should be able to access all required information concurrently, thereby avoiding unnecessary serial bottlenecks.

- *Locality of access* – for performance (on NUMA machines) and for reliability, scheduling information should be distributed across processors such that information is locally accessible whenever possible (e.g., local ready queues), thereby avoiding network contention and increased scheduling latencies due to global data accesses by dispatchers and scheduling tasks.

The algorithms presented in this paper avoid task migration for reasons explained in [15].

3 A Distributed Real-Time Multiprocessor Scheduler

3.1 Multiprocessor Scheduler Structure and Algorithm

Scheduler Structure. Clearly, the NUMA or distributed memory nature of large-scale parallel machines should determine the structure of multiprocessor schedulers. In our design, scheduling information is distributed to permit *concurrency* and *locality of access*, as mentioned in Section 2. Specifically, as shown in Figure 1, each processor has a local ready list, called an *Earliest-deadline List* (EL) [7], containing all tasks that are guaranteed to be locally schedulable. In addition, each processor maintains in its local memory a data structure, called a *Slot List* (SL) [7], that records the time intervals currently occupied on the processor.

Figure 1 shows the distribution of all scheduling information (SLs and ELs) across the different memory units of the parallel machine. It also shows how scheduling is performed given this distribution. Specifically, tasks performing dispatching and schedulability analysis do not run on some single dedicated processor. Instead, each processor has a local dispatcher accessing its local EL, and a copy of the scheduler's code for performing schedulability analysis. When a scheduler is executed on a processor, it uses the uniprocessor scheduling algorithm explained in [7] and the local SL for analysis concerning the task's scheduling on its own processor; it accesses remote SLs for analyses concerning task scheduling on remote processors. Our simple initial implementation of the distributed scheduler uses a single global task queue. This global queue, ordered by increasing deadlines, contains all tasks requiring schedulability analysis. Additional waiting queues may exist in individual processors for local arrivals, if desired [2]. As per the definition of dynamic scheduling, all tasks may arrive randomly in such queues during system execution. Note that realistic large-scale implementations will require that the single global queue be replaced by several concurrently accessible queues, at the possible loss of desirable properties in global orderings maintained among queue elements.

As evident from Figure 1, the distribution of scheduling information and the existence of a global task queue permit schedulers to execute on any of the p processors (numbered from 0 to $p-1$ arbitrarily) in the parallel machine. Since the global task queue may be simultaneously accessed by multiple schedulers, a mutual exclusion lock L is used to serialize all queue accesses. This ensures task scheduling in queue order (i.e., earliest deadline first). Furthermore, serialization of queue accesses by schedulers prevents the concurrent execution of multiple scheduler tasks, as shown in Figure 1 by the use of dashed ovals vs. solid ovals for depicting schedulers. As a result, no synchronization is necessary on the data structures accessed by multiple schedulers (i.e., the SLs distributed across the machine's p processors).

Scheduling algorithm. Whenever a scheduler runs, it first attempts to acquire the lock L on the global queue. After lock acquisition, it inspects the first task T in the queue and tries to schedule it on the 'best' processor, using the uniprocessor scheduling algorithm presented in [7]. According to our static multiprocessor scheduling algorithm described in [14], this 'best' processor should be the one on which T attains the maximal laxity value, which is determined by searching the SLs on all of the processors in the parallel machine. After T has been scheduled, it is removed from the waiting queue, and the lock L is released. The scheduler then finishes its current run. As stated in Section 2, task migration is not supported, so that each task has to be scheduled on a single processor. However, programmers may enable migration by explicit decomposition of a single logical task into several physical tasks, each of which may be scheduled separately by the scheduler.

Recall that the main subject of this paper concerns information sharing in the distributed scheduler. Such information sharing occurs in the scheduler during the selection of the 'best' processor for task scheduling. The remainder of this paper focuses on an evaluation of tradeoffs between algorithm quality vs. run-time latency experienced when different information is shared during 'best' processor selection. Toward this end, we call the multiprocessor scheduling algorithm explained

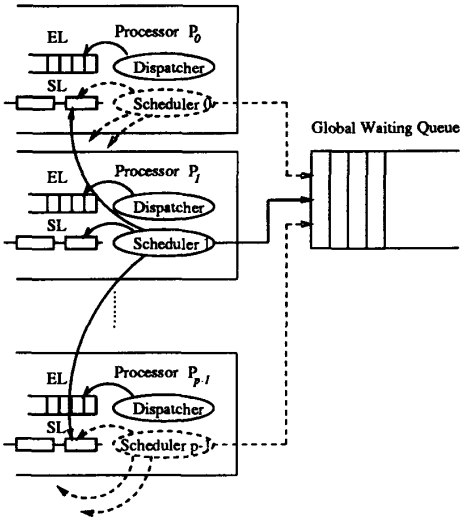

Figure 1: Structure of the Multiprocessor Scheduler

in this section an *SL-algorithm*, because it uses the complete remote scheduling information contained in processors' SLs for 'best' processor selection. Since this incurs significant run-time costs due to remote memory accesses, the SL-algorithm's latency is reduced when incomplete scheduling information is used, at some loss in quality of scheduling decisions. Accordingly, in Section 3.2, we introduce the *load interval* representation for incomplete scheduling information, and we define an *m-algorithm* as the variant of the SL-algorithm that (1) uses slot lists locally and (2) uses m load intervals as approximations of remote processors' scheduling information. Performance comparisons between these algorithms are presented in Section 4.

3.2 M-Algorithms Using Incomplete Scheduling Information

In the SL-algorithm, the SLs of p processors are the basis for sharing scheduling information among multiple schedulers. Specifically, whenever a scheduler needs to choose the 'best' processor for a task to be scheduled, it calculates the task's current p laxity values by searching these p SLs. The task is then scheduled on the processor with maximal laxity value. However, experimental results demonstrate that for NUMA architectures (and also for distributed memory machines), such searches incur significant costs due to remote memory accesses. As a result, reductions in algorithm latency require additional reductions in the number of remote memory accesses. In other words, the use of incomplete scheduling information is indicated.

One type of incomplete scheduling information used in previous work is the cumulative computation time of tasks in remote ready queues [9, 5]. Unfortunately, such an approximation of processor load is not sufficient for real-time applications due to their task timing constraints defined by arrival times, start times, and deadlines. Specifically, a processor's real-time schedule may consist of a set of occupied time intervals that are not adjacent (i.e., there may be an idle period between any two consecutive busy periods). Therefore, estimates of total processor load are not good predictors for task schedulability. This is shown in the example in Figure 2 (a). In this example, task T with $[A, S, C, D]$ = [0,60,50,140] can only be scheduled on processor P_1 due to the task's timing constraints and the available times on processors P_1 and P_2 (on P_1, time 40-135 is available). However, total cumulative load on processor P_2 is less than that on P_1, therefore causing the task's assignment to the wrong processor.

The representation of incomplete scheduling information developed in our work is called *load intervals*. In this representation, the entire time span (ETS) from system-start-time to system-end-time is divided into a number of time intervals of equal length[1]: $[I\_start_i, I\_end_i]$, $i = 1, 2, ..., m$, where m is the total number of intervals. Processor workloads are maintained for each load interval $[I\_start_i, I\_end_i]$, where each instantaneous workload is defined as the ratio of cumulative task computation time occupied in $[I\_Start_i, I\_End_i]$ divided by the total length of interval $[I\_Start_i, I\_End_i]$.

[1] The last interval may have a different length.

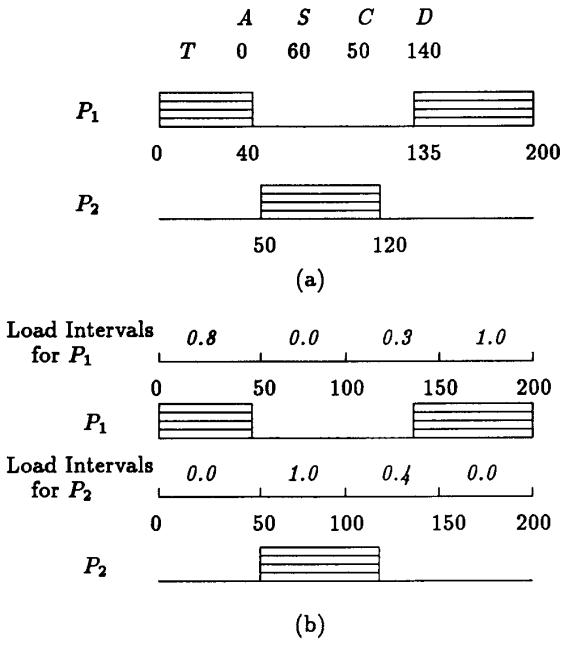

Figure 2: Example of the Load Intervals

A scheduler executing an m-algorithm uses the local SL only for schedulability analysis on its own processor, whereas load intervals are used to share scheduling information with other processors. Specifically, each processor maintains a *load array* data structure which records the workloads of load intervals on its own processor. Each array element represents a load interval $[I\_start, I\_end]$ with three fields: *interval_start*, *interval_end*, and *workload*. An m-algorithm selects the 'best' processor for task T_i as follows. First, it calculates a *containing period* for T_i. This period totally contains T_i's scheduling interval $[S_i, D_i]$ and it intersects some integer number of load intervals. Namely, the start time of the containing period is equal to the start time of the first load interval included in this containing period, and the end time of the containing period equals the end time of the last load interval included. Second, the m-algorithm computes the workload in the containing period for each processor from 0 to $p-1$, by accessing the remote load arrays. Task T_i is then scheduled on the processor with the least workload in the task's containing period, and the workloads of the appropriate load intervals on that processor are updated accordingly. However, it is not guaranteed that task T_i is schedulable on that processor. If it is not schedulable, task T_i is then scheduled on the processor with the second least workload in the task's containing period. If the task is again not schedulable, it is then scheduled on the processor with the third least workload, and so forth.

When using 4 such load intervals, the scheduling information in Figure 2 (a) appears as shown in Figure 2 (b), and a comparison of workloads in the task's containing period [50, 150] (which contains the second and third load intervals) correctly identifies processor P_1 as the 'best' processor.

Load intervals are easily varied approximations of scheduling information. The information carried by them is equivalent to the cumulative task computation time used in [9] when a single load interval spans all of the ETS on each processor. At the other extreme, the information carried in load intervals equals that contained in the SLs when the number of load intervals is equal to the total number of time units in the ETS.

4 Performance Evaluation

The following properties of the distributed scheduler are apparent from the previous sections:

- *Concurrency of access* – since scheduling information is separated into EL and SL lists, both of which are distributed across the processors in the

parallel machine, dispatchers resident on each processor are able to run concurrently with each other and with scheduling tasks performing schedulability analysis, thereby avoiding unnecessary serial bottlenecks.

- *Locality of access* – the distribution of ELs, SLs, and load arrays results in local accesses to scheduling information whenever possible, thereby also avoiding network contention and increased scheduling latencies due to remote memory references by dispatchers and scheduling tasks.

- *No task migration* – task migration is not permitted, so that the problems mentioned in Section 2 are eliminated. Specifically, (1) the costs incurred by task migration are avoided, and (2) processors are never forced to be idle when there are ready tasks.

The two remaining interesting performance aspects of the dynamic, distributed multiprocessor scheduler are (1) the performance effects of sharing complete vs. incomplete scheduling information and (2) scheduling performance in terms of the number of task sets that can be scheduled. We evaluate (1) by comparing the performance of m-algorithms with that of the SL-algorithm. Since there exists no optimal dynamic multiprocessor scheduling algorithm [3], (2) is evaluated based on the performance evaluation of our dynamic algorithm's static counterpart [14].

The results described in this section are attained by experimentation and simulation. Experimental results are attained on a sample NUMA machine, a 32-node GP1000 BBN Butterfly multiprocessor. Specifically, the uniprocessor and the distributed multiprocessor scheduling algorithms described in this paper are implemented using a real-time threads [8] package available on that machine, thereby permitting exact measurements of access latencies to local and remote data structures (i.e., ELs, local and remote SLs, and load arrays). In addition, scheduling latencies of the SL- and m-algorithms can be evaluated precisely. The quality of scheduling decisions made by SL- and m-algorithms is evaluated using synthetic workloads imposed on the scheduling algorithms. Such simulation runs are performed with the same algorithm implementation on a SUN Sparcstation.

4.1 Load Intervals versus Slot Lists

We evaluate the impact of information sharing in the distributed scheduler on two important factors regarding the performance of the scheduling algorithm: (1) the quality of scheduling decisions in terms of the number of task sets found schedulable and (2) the latency of scheduling decisions. Clearly, these two factors depend on the degree of completeness of scheduling information carried by the data structures being used. Below, we show that the quality of scheduling decisions improves as the degree of completeness of scheduling information increases (i.e., as the number of load intervals increases). However, scheduling cost (i.e., the latency of making a scheduling decision) also increases with the number of load intervals. Experiments are conducted to evaluate the actual delay caused by searching the load arrays versus the delay caused by searching the slot lists when a scheduler selects the 'best' processor for a task. Simulation studies are also performed to reveal the impact of the number of load intervals on scheduling quality. In addition, appropriate values for the number of load intervals corresponding to 'good' approximations of scheduling information are determined.

4.1.1 Quality of Scheduling Decisions

Ideally, the SL- and m-algorithms should be evaluated with synthetic workloads consisting of some number of task sets with feasible schedules. Algorithms' qualities can then be characterized as the number of task sets that can be feasibly scheduled by them. Unfortunately, the feasibility of a randomly generated task set can be determined only by exhaustive search. For p processors, the complexity of such a search for n tasks is $O(p^n * n!)$. Although branch and bound techniques may be used to reduce the search time, we consider such an approach impractical.

The synthetic task generator used in our simulation studies generates task sets with randomly generated start times, worst case execution times, and deadlines. 100 task sets are generated for each simulation run, given task set size, N, as an input. The task set size N determines system workload. All tasks in each task set are placed into the global waiting queue at the same time and are then scheduled one by one. Since each generated task set is not guaranteed to be feasible (i.e., a feasible schedule for the task set may not exist), scheduling quality of m-algorithms is compared with that of the SL-algorithm according to the number of task sets, out of the 100 generated sets, that are found schedulable. Task start time and task computation time are random variables with exponential distribution. The deadlines of tasks are defined as: deadline = start time + computation time + a random initial laxity value, where the initial laxity value is also exponentially distributed. Since simulation results using uniform start times and uniform computation times are found very consistent with the results using exponential distributions, they are not reported here.

For 24 processors, Table 1 illustrates the number of task sets that can be scheduled out of 100 by the SL-algorithm and by m-algorithms using various values of m. Table entries show scheduling results using varying workloads, i.e., using varying task set sizes. We assume that the ETS (entire time span) spans time 0 to 25000. The mean computation time of tasks is 100, tasks' mean initial laxity value is 1000, and tasks' mean start time is 2000. Note that light loads (e.g., $N = 100$) result in excellent scheduling decisions regardless of information completeness. Namely, the number of task sets found schedulable when $m = 1$ is almost identical to the number for $m = 600$. However, with heavier loads (e.g., $N = 700$ and above), significant differences in performance of the SL-algorithm compared to m-algorithms occur for varying values of m. Specifically, good approximations (high values of m) of complete scheduling information are required for high loads to approach the SL-algorithm's performance.

The results displayed in Table 1 are analogous to the results attained with parallel machines ranging in size from 4 to 128 processors. For comparison, some measurements using 64 processors are depicted in Table 2. Again, significant differences in algorithm performance appear for substantive loads.

In general, these simulation results demonstrate that increases in the value of m result in corresponding increases in the quality of scheduling decisions made by the m-algorithms. Furthermore, we conclude from these results that the incomplete information contained in the load intervals may replace the complete information in SLs even with relatively high system loads. Specifically, for system loads ranging from 100 to 600 tasks on 24 processors, the performance of the m-algorithms approximates the performance of the SL-algorithm for values of m exceeding 130 or 190. However, for real-time scheduling, it is not useful to share single values (i.e., $m = 1$) indicating processor load or utilization (as often done in distributed systems research). The utility of replacing the SLs with the load intervals is further corroborated by the measurements of actual scheduling decision latency described next.

4.1.2 Latency of Scheduling Decisions

Clearly, the scheduling quality decreases experienced by the m-algorithms compared to the SL-algorithm should be offset by improvements in scheduling latency. Theoretically, the time complexity of the SL-algorithm for searching p slot lists is $O(pn)$ in the worst case, where n is the number of tasks that have been guaranteed to meet their deadlines. However, since the number of slots in the SL is much less than n on average [7], the average complexity of SL searches is much better than the worst case time. Alternatively, when load intervals are used for representation of scheduling information, search time complexity is $O(p)$. However, the actual delay experienced by the m-algorithms' load array accesses also depends on the number of load intervals. This is because (1) the number of load intervals is inversely proportional to the load interval length for fixed values of the ETS and (2) short load intervals require more load intervals to be included in some containing period.

Slot lists and m-algorithm implementations on a 32-node GP1000 BBN Butterfly (a 68020-based machine) observe the following facts For slot lists, each slot access requires three memory references [7]: one for accessing the slot's address, a second for accessing the slot's start time, and a third for accessing the slot's end time. In contrast, each load interval in a load array can be accessed using a single memory reference, because array accesses are performed with indices replicated across processors' memory units and the number of array elements is fixed. This description might suggest that a ratio of 3:1 will result for the SL- vs. m-algorithms performance. However, this is not the case because all schedulers execute local code and access scheduling information on remote processors only during 'best' processor determination. As a result, actual differences in decision-making latencies between the SL-algorithm and the m-algorithms will be less than the ratio of 3:1.

Below, we refer to the process of selecting a 'best' processor as a *scheduling decision-making*. We obtain the scheduling decision-making latencies for a variety degrees of information completeness by measuring the scheduling delays caused by the SL-algorithm and the m-algorithms for different values of m. Figure 3 depicts the observed decision-making latencies for 24 processors as a function of system workload. At each load level, we measure the decision-making latency for each task in the task set. Note that the figure depicts the averages of measured values. Observed variances are small and are not shown in Figure 3.

The measured values depicted in Figure 3 demonstrate that decision-making latencies significantly depend on m, thereby demonstrating that it is important to obtain an acceptable degree of completeness in shared scheduling information. Furthermore, the use of incomplete scheduling information is shown preferable to the use of complete information, because the decision-making latencies of the m-algorithm for $m \leq 190$ are significantly lower than those of the SL-algorithm. This holds in conjunction with the simulation results in Table 1 where values of m equal to or exceeding 190 result in good scheduling quality for most reasonable system loads.

Another interesting result apparent from the data in

	Number of task sets found schedulable						
	m-algorithm						SL-algorithm
N	$m = 1$	$m = 130$	$m = 190$	$m = 300$	$m = 400$	$m = 600$	
100	99	100	100	100	100	100	100
200	96	100	100	100	100	100	100
300	89	100	100	100	100	100	100
400	75	100	100	100	100	100	100
500	54	98	98	98	99	100	100
600	39	93	96	96	99	98	100
700	23	87	88	90	94	95	99
800	8	63	76	81	83	83	90
900	4	42	53	61	65	66	72
1000	0	12	23	28	28	29	32

Table 1: Quality Comparison of Scheduling Decisions, $p = 24$

	Number of task sets found schedulable						
	m-algorithm						SL-algorithm
N	$m = 1$	$m = 130$	$m = 190$	$m = 300$	$m = 400$	$m = 600$	
1000	48	100	100	100	100	100	100
1200	38	98	100	100	100	100	100
1400	23	94	95	100	100	100	100
1600	12	80	89	99	100	100	100
1800	3	74	81	89	94	97	99
2000	1	50	62	72	78	87	92

Table 2: Quality Comparison of Scheduling Decisions, $p = 64$

Figure 3 is that slot lists are efficient representation of complete scheduling information for two reasons. First, when the load interval length equals unit time (i.e., the number of load intervals equals the total time of the ETS), the information carried in the load intervals is equivalent to the complete information contained in the slot lists. However, the latencies of the m-algorithms with $m = ETS$ substantially exceed that of the SL-algorithm. Second, with increasing system load, the latency of the SL-algorithm actually decreases due to slot merging and becomes lower than that of the m-algorithms with values of m exceeding 600 (and for values of m exceeding 400 for extremely high loads not shown in Figure 3).

4.1.3 Decision Latency for Distributed Memory Architectures

The ratio of local to remote memory access times is 1:7 on the GP1000 BBN Butterfly. An optimistic assumption places the corresponding ratio for distributed memory machines (e.g., the Intel iPSC series machines) at 1:50. We simulate expected performance on a distributed memory machine by experimentation with slightly altered implementations of the SL- and m-algorithms, where each remote memory reference (to a remote load interval or to a remote slot) is repeated 6 times. This results in an experienced ratio of local to remote memory access times of 1:42. In addition, we compare NUMA performance with UMA performance, using implementations of the SL- and m-algorithms where 'remote' data structures are actually resident in local memory on the GP1000 BBN Butterfly. Measurements are given in [15]. Those measurements demonstrate that load interval representations are not interesting for UMA machines, because such machines do not exhibit any differences in local to remote access costs. More importantly, those measurements demonstrate that the importance of incomplete versus complete information usage increases with increases in local to remote memory access costs [15]. We conjecture from our results that the sharing of complete scheduling information is not practical in distributed memory systems.

4.2 Performance of the Dynamic Multiprocessor Algorithm

To our knowledge, no optimal dynamic real-time multiprocessor scheduling algorithm [3] has been described in the literature. Furthermore, the generation of feasible task sets is an NP-hard problem. Therefore, we gain additional confidence in the performance of the SL- and m-algorithms based on the evaluations of the SL-algorithm's static (off-line) counterpart.

The static SL-algorithm also uses SLs and ELs to record scheduling information. It schedules tasks one at a time (as in the dynamic algorithm) and does not permit task migration. It differs from the dynamic algorithm only in its choice of the next task to be sched-

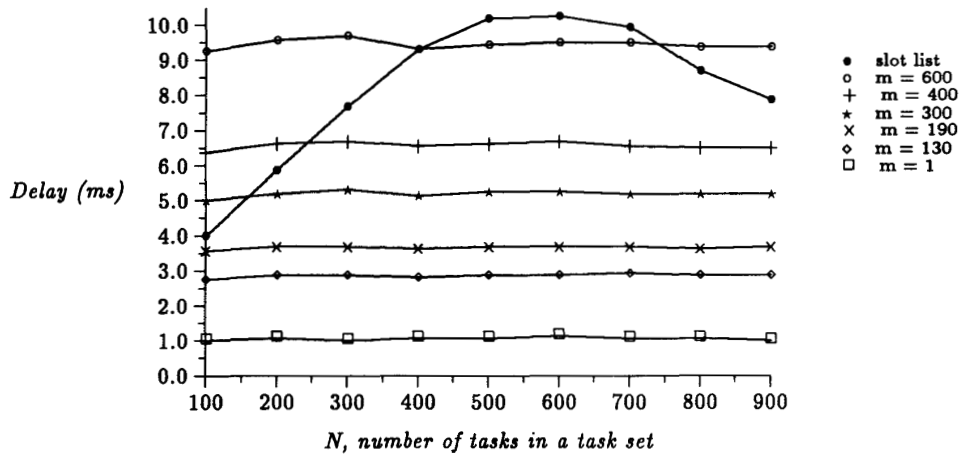

Figure 3: Comparison of the Average Scheduling Decision Latency

uled from the global task queue. Namely, as with the dynamic SL-algorithm, it will usually schedule tasks in order of deadlines, earliest first. However, in contrast to the dynamic SL-algorithm, tasks with zero or very small maximal laxity values are always scheduled first, even if tasks with earlier deadlines exist. Once a task has been chosen, it is scheduled on the processor on which it attains the maximal laxity, as with the dynamic SL- and m-algorithms in Section 3.1. Therefore, the static SL-algorithm is basically an EDF algorithm without task migration. However and in contrast to other pure EDF algorithms, the static SL-algorithm uses tasks' laxities to discover those tasks that must be scheduled immediately. This is useful when there are several tasks with relatively large computation times (and large deadlines) but small laxities.

Details of the static SL-algorithm, examples, and a performance comparison to a static algorithm using a pure LLF scheme (a simplified version of the algorithm in [13]) appear in [14]. In that evaluation, the static SL-algorithm's performance is shown competitive to that of the LLF algorithm in terms of the number of task sets that can be scheduled. By formal proof, we further show that the static SL-algorithm is better than a pure EDF algorithm, and by simulation, we demonstrate that the static SL-algorithm performs as well as the LLF algorithm when the cost of task migration is assumed 5% or more of the average task computation time.

5 Conclusions and Future Work

The two contributions of this paper are: (1) the design and implementation of a distributed multiprocessor scheduler and (2) the evaluation of a characteristic of this scheduler important to most parallel or distributed programs, which is:

> What are the performance effects of information sharing within a distributed scheduler?

Specifically, we show that the sharing of incomplete (vs. complete) information within the distributed scheduler can lead to increased performance regarding scheduling latency with few or no losses in scheduling quality. In addition, we show that this holds for a variety of parallel machines, ranging from NUMA to distributed memory machines, where differences in performance due to information sharing are exacerbated on target parallel machines with increased latencies of access to remote information.

We also develop a novel and flexible representation for shared information within a distributed, real-time scheduler – termed *load intervals*. This representation and the scheduling algorithm being employed can be easily changed regarding the degrees of completeness of information sharing (and therefore, the quality vs. latency of scheduling decisions), so that the resulting distributed scheduler can be easily scaled and adapted to different large-scale parallel machines.

Our future work concerns the implementation of realistic distributed operating system services, including: (1) the scheduling of groups of related tasks that must be run in parallel or on subsets of processors, (2) the effects of concurrency of scheduling on scheduling latency and quality, (3) the use of scheduling strategies other than 'best first' when selecting a processor for task scheduling.

References

[1] David L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[2] Ben Blake and Karsten Schwan. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*, 17(1):34–44, Jan. 1991.

[3] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

[4] W. A. Horn. Some simple scheduling algorithms. *Naval Res. Logist. Quart.*, 21:177–185, 1974.

[5] L. M. Ni and K. Hwang. Optimal load balancing in a multiple processor system with many job classes. *IEEE Trans. on Software Engineering*, 11(5), 1985.

[6] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami, Florida*, pages 22–30. IEEE, Oct. 1982.

[7] Karsten Schwan and Hongyi Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[8] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Multiprocessor real-time threads. *ACM Operating Systems Reviews*, 25(4), October 1991.

[9] K.G. Shin and Y.C. Chang. Load sharing in distributed real-time systems with state change broadcasts. *IEEE Trans. on Computers*, 38(8), August 1989.

[10] John A. Stankovic and Krithi Ramamritham. The spring kernel: A new paradigm for real-time operating systems. *A Quarterly Publication of the Special Interest Group on Operating Systems*, 23(3):54–71, July 1989.

[11] Jia Xu and David Lorge Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3), March 1990.

[12] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *SIGMETRICS' 90, Boulder, Colorado*, pages 214–225. ACM, May 1990.

[13] Wei Zhao, Krithi Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, August 1987.

[14] Hongyi Zhou and Karsten Schwan. Real-time multiprocessor scheduling using both deadlines and laxities. Technical report, College of Computing, Georgia Institute of Technology, Atlanta GA, September 1991.

[15] Hongyi Zhou, Karsten Schwan, and Ian Akyildiz. Performance effects of information sharing in a distributed multi processor real-time scheduler. Technical report, College of Computing, Georgia Institute of Technology, GIT- CC-91/40, Sept. 1991.