Rapid – A Multiprocessor Scheduler for Dynamic Real-Time Applications

Harold Forbes

Karsten Schwan

College of Computing

Georgia Institute of Technology

Atlanta, GA 30332

April 30, 1994

Abstract

This paper describes and evaluates operating system support for on-line scheduling of real-time tasks on shared memory multiprocessors. The contributions of this work include: (1) the design and implementation of an efficient on-line scheduler that can execute a variety of policies addressing both the assignment of real-time tasks to processors and the scheduling of tasks on individual processors, (2) performance improvements in multiprocessor scheduling due to the separation of task schedulability analysis from actual task scheduling and due to the use of parallelism internal to the scheduler, and (3) the scheduling of individual as well as sets and groups of tasks. Performance measurements on a multiprocessor machine describe the costs and benefits attained from (2) and (3), based on experiences with a multiprocessor robot navigation and planning program being implemented as part of this research.

1 Runtime Support for Real-Time Systems

Complex real-time applications. Most modern embedded architectures consist of multiple processors operating in parallel to achieve the high throughput and low latency required by their application software. Accordingly, such software is partitioned into multiple tasks that potentially execute concurrently, often using shared resources, and typically requiring that task-specific timing constraints be met during execution. Moreover, real-time systems are inherently *dynamic* when they operate in complex external environments or consist of many interoperating, asynchronous, and potentially distributed code modules. External events causing dynamic system behavior include: (1) unforeseen events, like the detection of new obstacles in robot navigation[2], (2) actions by human operators, and (3) unexpected system loads due to excessive levels of inputs in certain sensors, such as the arrival of threats in military applications[15]. In addition, internal characteristics of real-time computer systems leading to dynamic system behavior include software and/or hardware faults due to the system's inherent complexity[5] and temporary actions required by specific system components, to avoid component damage, to trade off component performance vs. reliability, etc.

The `Rapid` scheduler. This paper presents the `Rapid` scheduler for dynamic real-time, multiprocessor systems. This scheduler is designed to support autonomous robotics application, where single or multiple robots operate in potentially unknown environments and where robot application software is implemented as a group of cooperating and communicating threads jointly serving robot sensors and actuators and sharing the resources of the robot's embedded computer system. As a result, the `Rapid` scheduler has to support the on-line scheduling of time-constrained individual tasks, sets of unrelated tasks, and groups of tasks cooperating to solve a single problem. Furthermore, dynamically arriving tasks may be sporadic or sporadically periodic, the latter requiring their periodic execution for some limited amount of time. Task sets or groups may consist of any number of sporadic or periodic tasks. Task arrivals may occur at any time during system operation, and task time constraints may be characterized by soft or hard deadlines and start (or release) times known at the time of task arrival.

On-line real-time scheduling. On-line real-time schedulers have been constructed in many past and several recent research efforts. The Spring operating system designers have implemented and evaluated multiprocessor schedulers running on single, dedicated nodes of small-scale parallel embedded systems[25], with recent work addressing hardware support for on-line scheduling[16], and past work addressing distributed real-time systems[18]. Shin et al.[13] and the HartOS operating system group has experimented with tradeoffs in communication vs. performance and quality in distributed real-time scheduling. In addition, research at Carnegie Mellon University has been extending priority-based scheduling methods to address dynamic system behaviors, typically by development of novel scheduling algorithms[24]. Multiprocessor implementations of internally concurrent schedulers were first described in [7], then generalized and evaluated rigorously and experimentally in [23, 30]. The latter work also developed new algorithms for schedulability analysis and addressed the scalability of schedulers in terms of their locality characteristics on large-scale parallel machines. Also, higher level mechanisms for managing soft real-time parallel applications are being at the University of Rochester[28].

Scheduler performance and concurrency. The `Rapid` scheduler builds on our previous research in real-time,

multiprocessor threads[30], but focusses on two topics not explored to date: (1) the use of *concurrency* during scheduling and (2) the *efficient implementation* of concurrent multiprocessor schedulers on large scale parallel machines. This research currently uses a 64-node shared memory KSR-1 multiprocessor; its primary contributions are:

- **Scheduler concurrency** – in contrast to [30], schedulability analysis and scheduling are performed by multiple, concurrently executed threads, thereby enabling us to vary both the latency of scheduling for individual requests and scheduling throughput, by variation of internal scheduler concurrency. Presentation of performance gains due to parallelism appear in Section 4. Tradeoffs in performance gains vs. the resulting quality of multiprocessor scheduling are shown to favor efficiency over additional efforts to increase quality.

- **Asynchronous scheduler execution** – the `Rapid` scheduler *decouples* (1) the *generation* of scheduling requests associated with new task arrivals, from (2) the *schedulability analysis* performed for such requests, from (3) the *scheduling* of tasks. The associated program interface presented by `Rapid` and its use by a sample robot navigation program are presented in Section 2 of this paper. The internal scheduler mechanisms used for the decoupled execution of steps (1)-(3) are described in Section 3. Performance advantages derived from the resulting asynchrony among steps (1)-(3) are evaluated in Section 4.

In future work, alternative implementations of internal interactions among different scheduler components will be evaluated to address the general topic of suitable operating system interfaces for real-time operating systems' resource managers[1].

The remainder of this paper is structured as follows. First, a sample multiprocessor application is shown to require on-line real-time scheduling. This robot planning and navigation code motivates the functionality and demonstrates the interfaces offered by `Rapid`. In Section 3, the internal structure of `Rapid` are described in detail, in order to explain the performance measurements in Section 4. Conclusions and future research appear in Section 5.

2 `Rapid` Scheduling for Robot Navigation and Planning

2.1 Schema-based Robot Navigation

The sample application used for evaluation of `Rapid` is a parallel code performing robot navigation to a known goal position. Navigation is performed across an un-mapped world potentially cluttered with obstacles. The parallel code is based on the reactive component of the Autonomous Robot Architecture (AuRA) [3], in which motor schemas are the basic unit of behavioral control of the physical system. Concurrency of execution is possible because several schemas may be active simultaneously as the robot moves. Schemas jointly generate an overall navigational behavior: (1) perceptual schemas process sensor data and provide inputs to, (2) motor schemas that generate movement vectors, and (3) movement vectors are summed and normalized before the result is transmitted to the robot's actuators (or simulated robot) for execution of movement. Additional schemas are responsible for planning, using a parallel implementation of a planning algorithm similar to the D* algorithm by Stentz[27].

Motor, perceptual, and planning schemas are implemented as concurrent execution threads with the Cthreads library for parallel programming [21]. A basic navigational strategy requires threads for: (1) `sensor`, which provides sensor inputs and also terminates processing when the robot reaches the goal, (2) `move-to-goal, noise, and avoid obstacle`, which are each single threaded motor schemas, and (3) `move-robot`, a single thread which references the outputs of the motor schemas and effects robot movement. While the behavior of this set of schemas is robust (ie., the robot will not run into obstacles), the robot may fail in complex environments (e.g., it can become trapped in box canyons). Complex environments require additional schemas, including a `planner`, which has multiple threads jointly computing a plan based on which successful movement toward the goal can be ensured. The task of `planner` threads is to find an optimal (for the known world) route to the goal, and their continuous execution at some level of criticality is indicated, in part to minimize delay upon discovery of new obstacles by basing new plans on partially and continuously computed alternate plans.

Timing constraints in the robot application are due to the operational requirements of the underlying vehicle, such as its sensor's ranges and rates of output and its actuators' required rates of control. In addition, thread execution periods directly depend on vehicle speed, which in turn depends on the computational resources available to execute the threads. Unexpected events result in dynamic changes to threads' timing requirements:

- Unforeseen obstacles detected during vehicle movement require re-planning. This means that previously unscheduled `planner` threads must be created and scheduled or that increased attention is given to previously scheduled `planners`, therefore requiring their rescheduling.

- As in other autonomous systems[26][17][20, 9], fail-safe plans are used to preclude abrupt mission termination, excessive waiting, or failure in unknown environments. Therefore, the detection of a new obstacle not only may interfere with the existing plan and therefore, cause re-planning, but it also triggers a fail-safe strategy that reduces the vehicle's speed to a guaranteed safe level and causes the vehicle to continue moving in the general direction of the goal while avoiding obstacles. This may be implemented by dynamic creation and scheduling of a sporadic thread executing the fail-safe plan.

Dynamic obstacle detection, continuous and concurrent planning and re-planning, and fail-safe robot operation jointly imply that execution threads must be scheduled dynamically, during system operation, and that scheduling must be performed such that thread timing constraints can be easily varied. In fact, experimentation with a prototype of the concurrent robot code described in [4] demonstrate both the importance of concurrency in thread execution and the need for dynamic control of thread scheduling:

- 'best effort' thread scheduling disregarding thread timing constraints demonstrates the existence of parallelism in the robot code, but

- also shows that the dynamic variation of threads' timing constraints can significantly improve the speed and quality (measured as total path length from origin to goal) of robot movement when schemas' rates of execution are continuously adjusted according to their current criticality.

A suitable application interface for dynamically schedulable and re-schedulable threads, with varying time constraints, is described next, using fragments of the robot application discussed above.

2.2 Rapid Application Interface

Idealized schema execution. The outline of the operation of a generic robot schema is:

- run schema code,
- calculate next invocation parameters, and
- reschedule self.

Ideally, scheduling is performed for each execution of a schema instance. This is because schemas are designed to control highly reactive autonomous systems like 'cockroach' robots, unstable flying vehicles like helicopter, etc. In addition, schemas are often relatively 'small' in terms of the number of instructions executed per schema run. The resulting frequent need for schema scheduling and rescheduling is one of the primary motivations for the design and implementation of the `Rapid` scheduler, which uses the robot navigation code described in this section as one instance of a real-time code 'stressing' its on-line scheduling technology.

Dynamic periodic schemas. While an idealized schema requires scheduling for each execution, in practice and with the particular vehicles used in our research (a Denning mobile robot at present, and an autonomous wheeled vehicle in the future), scheduling overheads may be reduced by scheduling several periodic executions of each schema at one time. Essentially, schema instances are executed as periodic tasks with limited time horizons, as shown in the code below:

```
void schema()
{
  run_schema_code();
  current_invocation++;
  if (current_invocation = horizon) {
    current_invocation = 0;
    horizon = calculate_horizon (obstacles);
    period = calculate_period (vehicle_speed);
    if (!RTasync_multi_fork (schema,NOW+period,runtime,period,
                             horizon,slow_and_reschedule))
      slow_and_reschedule();
  }
}

int calculate_period (vehicle_speed)
```

```
{
  return ((sensor_range/vehicle_speed)/safety_factor);
}

void slow_and_reschedule()
{
  vehicle_speed = vehicle_speed - (vehicle_speed/backoff_factor());
  horizon = calculate_horizon (obstacles);
  period = calculate_period (vehicle_speed);
  if (!RTasync_multi_fork (schema,NOW+period,runtime,period,
                           horizon,slow_and_reschedule))
    slow_and_reschedule();
}
```

In this code, `Calculate_horizon()` determines the number of periods that can be executed using the current vehicle parameters and known obstacles. Open areas with few obstacles generate longer horizons, while close obstacles and high obstacle densities generate shorter horizons. In addition, `Calculate_period` ensures that for a given sensor range and the current vehicle speed, the vehicle cannot impact an obstacle that is just beyond its sensor range for the duration of each period.

The `Rapid`-provided scheduling construct `RTasync_multi_fork()` schedules a `horizon` number of schema invocations beginning at time `NOW+period`, with a return value indicating success or failure. For low latency, this call internally performs several levels of schedulability analysis. At the first level, 'sanity checks' concerning the specific scheduling request and crude estimates of available execution times are used to indicate the probable success or failure of the submitted request, so that upon failure, the robot code can quickly take some alternate action (discussed next). When success is probable, the `Rapid` call initiates thorough schedulability analysis and returns the value `TRUE`, eventually resulting in the newly created set of tasks (schema instances) being executed before their desired time horizon (interpreted as a hard deadline by `Rapid`). If scheduling is not likely to succeed, `FALSE` is returned causing the recursive execution of alternate actions, in this case consisting of execution of the procedure `slow_and_reschedule()`.

In `slow_and_reschedule()`, a lower vehicle speed increases the period, thereby reducing the total computation time needed before the given time horizon. `slow_and_reschedule()` internally attempts to schedule the resulting schema instances at their new periods. Last, the procedure `Backoff_factor()` will halt the vehicle before `period` time has passed and before the schema's deadline is reached. This corresponds to an emergency reaction and may take other forms in different real-time applications, such as increases in altitude in autonomous guided missiles, the reversion to human intervention in the ASV walker[6] or in other semi-autonomous vehicles, etc.

Sporadic schemas. While the previously described schemas can gradually adjust vehicle speed and direction in response to environmental changes, the detection of a new obstacle may also require an immediate and significant

change to vehicle movement, such as a complete course reversal (e.g., consider 'cockroach' vehicles driven by schema-based navigation). Consequently, when the `sensor` schema detects a previously unknown obstacle (code not shown for brevity), it takes two actions: (1) it executes the `Rapid` call `RTsync_fork(new_obstacle,NOW,runtime,deadline)`, and (2) it switches from the `follow-plan` schema to the slower but more robust `move-to-goal` schema. It is important to note that the execution of `new_obstacle` is not a safety requirement, since `avoid-obstacle` will continue to ensure that the vehicle does not crash into any obstacle. Therefore, the `deadline` used in the `fork` instruction simply needs to ensure that schema `new_obstacle` is scheduled sometime before the obstacle is reached. Upon execution of `new_obstacle`, if the newly detected obstacle does not actually block the current path, then the vehicle reverts to following its previous plan and no vehicle parameters are changed. Otherwise, `new_obstacle` makes a heuristic decision about how significant the re-planning effort will be, may reduce vehicle speed to permit time for re-planning and/or to free computational resources, and forks additional planning threads.

`Planner` schemas. Planning schemas are scheduled, run, and re-scheduled like motor schemas. However, since their continuous execution is desired rather than essential, if their re-scheduling fails, then planning periods are simply increased without altering vehicle operation. In addition, planning threads autonomously increase their periods when no additional planning has to be done due to the lack of new obstacles or of other changes in the robot's external environment. When periods exceed some pre-determined value, planning threads are not re-scheduled. Conversely, planning threads with excess work will attempt to reduce their periods and thereby, increase their total execution time before their planning horizons.

2.3 `Rapid` Functionality

`Rapid` call interface and timing model. We have described the operation of `Rapid` using (1) idealized, constantly re-scheduled schemas and (2) sporadic and dynamic periodic schemas with limited time horizons. Specific examples of schemas are periodic planning schemas performing self-rescheduling and sporadic fail-safe schemas performing `move-to-goal` processing rather than following an outdated plan. The `Rapid` scheduling calls used for such schemas are: `RTsync_fork(new_obstacle,NOW,runtime,deadline)` and `RTasync_multi_fork()`, which create and schedule either a single thread synchronously with the execution of the requester or a set of threads asynchronously with the requester's execution. The thread's timing constraints are described by parameters (A, S, C, D), with A being its arrival time, S being the earliest possible time at which its execution may begin (start time), C being the estimated maximum computation time, and D being the deadline by which it must complete its execution, none of which need to be known until the time of thread's scheduling (in this case, until its creation). The *laxity* l of thread 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 thread's laxity changes over time. A thread's laxity may be used as a measure of its urgency at some given point in time. For example, a thread's laxity of 0 at time t implies that the thread must be scheduled immediately in order to meet its deadline. A thread's *maximum laxity value* is the maximal value among its set of laxities on the parallel machine's processors. `Rapid` uses task laxity to order scheduling requests.

In addition to dealing with timing constraints, `Rapid` calls also offer return values and exceptions concerning

scheduling (e.g., to indicate that re-scheduling fails and must be retried). In addition, and more importantly, two different types of calls are offered: (1) synchronous calls (ie., `RTsycnch-fork`) and (2) asynchronous calls (ie., `RTasynch-fork`). Synchronous calls imply that the caller waits until scheduling has been performed, whereas asynchronous calls permit the caller to continue (perhaps performing other tasks) while tasks are being scheduled. Asynchrony is discussed in more detail below, since it is a prerequisite for attaining concurrency in task scheduling.

Scheduler functionality. Given the timing model and scheduling calls described above, the functions of the `Rapid` scheduling algorithm are *schedulability analysis* and *schedule construction*. Schedulability analysis determines whether a feasible schedule exists for a single or set of tasks, whereas schedule construction calculates a feasible schedule, if it exists. A schedule is *feasible* if all tasks in the set can be assigned to processors and scheduled such that their timing constraints are met. A set of tasks is *schedulable* if there exists at least one algorithm that can feasibly schedule the set. 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. Note that two important characteristics of this definition are (1) that all scheduling guarantees are maintained after they are made (any other assumption would lead to unreasonable costs and lack of predictability in actual real-time systems), and (2) that schedulability analysis and schedule construction are separated. This separation permits the rapid recognition of failures regarding task scheduling, so that the application program or higher-level operating system software [10, 12] can deal with such failures in a timely manner (e.g., by submission of alternate tasks, by reduction of execution time using alternate algorithms, etc.).

Schedulability algorithm. `Rapid` uses Zhou's slot list algorithm to perform schedulability analysis of a task on each processor. This algorithm takes $O(n \log n)$ steps to determine the schedulability of any new task, where n is the number of slots in the slot list. Detailed studies of average slot list lengths indicate that algorithm performance remains good and even improves with increasing system loads[23]. However, in contrast to `Rapid`, the multiprocessor slot list-based algorithm described in [30] only allows a single scheduler to be active at a time. Therefore, although actual scheduling overhead is comparatively low, overall scheduling latency could become unacceptable in the presence of a large queue of tasks to be scheduled. The `Rapid` concurrent scheduler allows multiple scheduling threads to be active simultaneously thereby, increasing throughput and decreasing latency.

The execution of `Rapid`. The `Rapid` scheduler is not simply run as independently scheduled threads with fixed execution periods staggered across different processors, as the non-concurrent multiprocessor scheduler described in [30]. Instead, whenever a schema finishes its execution, the low-level task dispatcher within `Rapid` can choose to run either a thread executing the scheduler or the next available application task. The scheduler is run if (1) there is sufficient time to run the scheduler at least once before the next task's scheduled start time, and (2) if no other processor is capable of analyzing the currently outstanding request(s). Otherwise, the dispatcher will execute the next available task that has already passed its start time, even if that task was originally scheduled to begin its execution later. In summary, `Rapid` scheduling is performed opportunistically whenever processing time is available on any of the processors capable of analyzing a given set of tasks (called a *scheduling group* in Section 3). However, for reasons of system predictability under high system loads, `Rapid` schedulers will also be scheduled conservatively

using a dynamic variant of the staggered scheme described in [30].

2.4 `Rapid` Concurrent Scheduling

To attain concurrency in task scheduling, `Rapid` must decouple (1) the generation of scheduling requests, from (2) schedulability analysis for requests, from (3) the decision concerning thread scheduling. Orthogonally, thread creation itself can also be decoupled from its scheduling, by using the `RT-thread-prefork` calls described in [22] (not used in this example), or by first creating a thread with a non-real-time fork instruction and then scheduling it using `RTasynch-schedule()`. The primary mechanism supplied by `Rapid` for attaining asynchrony and to permit thread creation separately from thread scheduling is the *reservation*. Reservations and the decoupling of (1)-(3) are important for two reasons:

- *Concurrency* in schedulability analysis implies the use of multiple, potentially asynchronous scheduling threads in order to analyze a single scheduling request. A *reservation* associates the requesting entity (some thread calling `RTasynch-fork`) with threads performing schedulability analysis and scheduling. The requesting thread can proceed asynchronously with such threads, or it can wait for analysis to be performed.

- *Flexibility* and therefore, suitable performance require the potential use of different policies for decision-making concerning thread scheduling (e.g., best vs. any fit across different processors), of different degrees of parallelism in schedulability analysis (resulting in different execution rates for threads performing such analyses), and of different means for accessing and updating scheduling requests. In response to these needs, `Rapid` offers several levels of decision making concerning scheduling requests and multiple means of storing and sharing reservations among scheduling threads. It also allows a requesting thread to wait on and make decisions based on its reservation, and it can indicate in each reservation multiple, potentially suitable time slots on different processors.

Therefore, in contrast to previous work on *scheduler activations* described in [1], the role of a *reservation* in `Rapid` is to provide a vehicle for maintaining information about specific scheduling requests, for single or sets of tasks, and to permit application threads and the different components of the `Rapid` scheduler to cooperate (synchronize) and communicate (exchange information) to satisfy such requests.

Multiple levels of decision-making. As stated above, another difference of `Rapid` scheduling to previous work is the immediate feedback given to a requester at the time of request generation. As a result, a requester can make a heuristic decision on whether schedulability analysis is likely to be performed in time and/or is likely to succeed, given the known rates of execution for threads performing schedulability analysis and given current knowledge of likelihood of success in scheduling. If the request is likely to fail, then no reservation is generated and the scheduling request returns a failure indication. As a result, the more time-consuming reservation generation, queueing, and schedulability are performed only when there exist some hope of success for the request. Since even probably successful scheduling requests may fail, the requester can also supply a *failure function* with the initial request which is included with the reservation. If the request fails, the scheduler itself (rather than the requesting task) immediately

executes this failure function. Alternatively, failure can be detected and acted on when the requester inspects the return parameters of the submitted request.

Additional support for handling scheduling failures must either be part of higher level real-time operating system functionality or must be supplied by application programs. One approach to implementing such support is taken by the CHAOS^{arc} is an object-based, real-time operating system kernel[11], where related groups of object invocations may be scheduled individually but can then be grouped together into an *atomic computation*, which is viewed by the operating system as a single schedulable unit with guaranteed scheduling, consistency and failure recovery attributes.

3 The `Rapid` Scheduler

3.1 A Scalable Implementation

NUMA machines. The previous section demonstrated the basic functionality of the `Rapid` on-line real-time scheduler. This section describes the scheduler's concurrent and distributed implementation on a shared memory multiprocessor. While our current platform is not an embedded machine, it has several characteristics in common with most embedded parallel architectures:

- Machine scalability implies non-uniformity in access times to shared memory (NUMA), resulting in memory access times to cache, local, and remote memory units differing by factors of roughly 1:10:100. Such properties are shared by most embedded architectures, with larger differences in access cost existing when message communications must be used in place of shared memory accesses.

- Devices may be attached to any one node of the parallel machine, and operating system kernels reside on each of its processors, while higher level operating system utilities run as processes on any number of nodes. As a result, real-time tasks can be created on any node of the parallel architecture, and they can run on any or a set of nodes possessing their devices (or can remotely use devices for low-bandwidth accesses).

As with most parallel machines, to attain speedup in parallel program execution, KSR programmers are able to reserve processors for their own use, thereby enabling us to experiment with user-level support for real-time scheduling not subject to arbitrary interference from the current machine's non-real-time operating system.

Concurrent execution, distributed data structures, and locality of access. The `Rapid` on-line scheduler is structured to attain high performance on NUMA machines of varying sizes. First, to attain locality of reference a dispatcher and a run queue – called *task list (TL)* – are located on each processor of the machine. In addition, a data structure maintaining higher level information about current processor schedules – called *slot list (SL)* – is also located on each processor, so that a local scheduler can perform its own schedulability analysis and scheduling without requiring access to non-local information. Second, in contrast to our previous work described in [30], `Rapid` supports concurrency in scheduling by decoupling (1) the generation of scheduling requests (generating a *reservation*), from (2) their processing (schedulability analysis by slot list inspection), from (3) task scheduling (task insertion into some TL). Third, `Rapid` implements several levels of schedulability analysis, so that requests being submitted

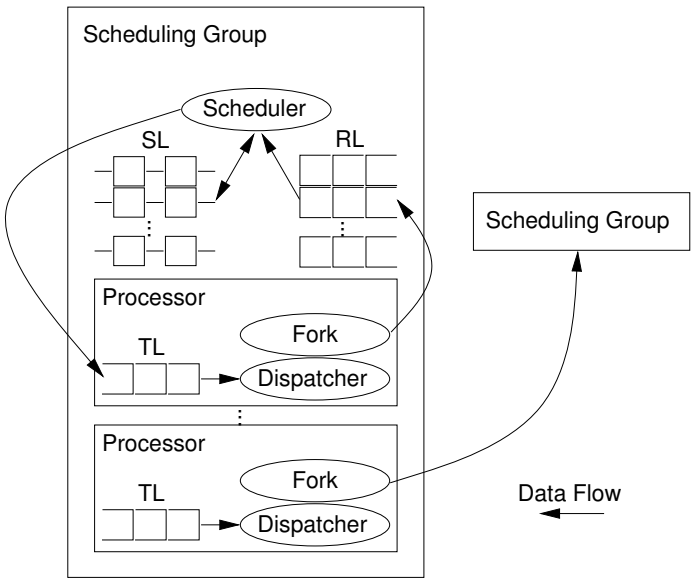

Figure 1: Structure of the Multiprocessor Scheduler

can be (a) quickly diagnosed as reasonable or unreasonable, (b) analyzed to provide hard deadline guarantees, and (c) analyzed to provide hard deadline guarantees by also re-scheduling previously scheduled tasks. However, such re-scheduling will never be performed such that previously made guarantees are retracted.

Reservations and reservation lists are the mechanisms supporting the decoupling of (1)-(3). Specifically, and as shown in Figure 1, when the currently executing task finishes, the dispatcher selects the next task to execute from its *task list* (TL), which involves neither remote memory accesses nor the creation of manipulation of reservations. However, when an executing task *forks* a new task or re-schedules an existing task, then `Rapid` creates a *reservation* for the task, based on which schedulability analysis and scheduling may be performed. The reservation is inserted into a *reservation list (RL)*, which is structured to match the machine's NUMA characteristics and can accommodate the new task's constraints concerning processors and/or devices required for its execution.

Reservations are inspected by threads performing schedulability analysis, and are removed from reservation lists only when schedulability analysis is completed. Schedulability analysis is performed using Zhou's[29] slot list algorithm, where successful analysis results in the generation of a *reservation entry* by the scheduler into the reservation maintained for this task.

A scheduling decision is made for each reservation as soon as permitted by the decision-making algorithm being used, which may be 'best fit' (requiring that all schedulers inspect the task) or 'any fit' (allowing the 'first' successful scheduler to accept the task for insertion into its TL), etc. Once a decision has been made, the reservation is removed from the reservation list and the task is inserted into the appropriate task list. Since such decision making and task list insertion are performed by the threads performing schedulability analysis, locality of reference is maintained even when executing user-provided *failure functions* performing exception handling upon failure. Reservation entries for this task not accepted are removed by scheduling threads, thereby again maintaining locality of access to slot lists.

Variation of scheduler concurrency. The level of concurrency during schedulability analysis can be varied by creation of any number of threads performing this analysis for single or multiple processors, thereby resulting in a single thread manipulating some fixed number of slot lists corresponding to the number of processors for which it is responsible. Strict locality requirements indicate the need for one scheduling thread per processor, so that only the reservation list is shared between multiple processors. It is possible to relax such requirements on machines like the KSR, where a moderate number of remote memory accesses are possible due to automatic caching performed by the underlying machine's architecture. Operationally, scheduler configuration resulting in changed locality and concurrency levels for scheduling is performed using *scheduling groups*, where in Figure 1 a single scheduling thread accesses an internally distributed reservation list (RL) and manipulates a limited number of SLs and TLs on behalf of each reservation being serviced.

Since any number of threads performing schedulability analysis can be run concurrently, improvements in scheduling latency are attained when a single task's schedulability is evaluated for multiple target processors and when sets of tasks are evaluated. Performance effects due reservation and reservation list implementation, tradeoffs in scheduling performance vs. quality of scheduling decisions, and improvements in scheduling latency and throughput are evaluated in Section 4.

Scheduler configuration. The flexibility of `Rapid` is enhanced by describing the scheduler's desired configuration with an explicit configuration description. This also permits the dynamic change of scheduler structure to adjust it to different machine configurations and runtime needs. Furthermore, a processor's membership in a scheduling group is defined explicitly in each scheduler's state. This permits processors to be dynamically added to or removed from scheduling groups, and processors can simultaneously belong to more than one scheduling group. The size of scheduling groups results in tradeoffs in scheduler performance vs. the quality of scheduling exposed in part by our past work [30] and to be investigated further in our future research. Similarly, communication between processors and the scheduler is via the per-processor reservation list (RL) part of each scheduler's state. Experimentation with shared RL's has shown poor performance due to the high remote/local memory access ratios resulting from such RL sharing (see Section 4).

3.2 Performance of Concurrent Scheduling

Figure 2 depicts the decoupled execution of task generation (the 'fork' instruction), schedulability analysis, decision making, and task execution, involving several processors in the same and in different scheduling groups. Processors *p0*, *p1* and *p2* constitute scheduling group 1(*sched_group1*) and processors *p3*, *p4* and *p5* constitute scheduling group 2 (*sched_group2*). Normally, a task created in a scheduling group will be executed by one or more of the processors in that scheduling group. For instance, *fork1* on p0 generates a reservation for *task1* that is scheduled by *scheduler1* on p2. However, when tasks have processor constraints or when information is available to the fork procedure that schedulability analysis will not succeed in its own scheduling group, then the fork will employ the reservation list to submit the reservation to a different scheduling group. For example, *Fork3* in sched_group2, recognizes that *task3* has a deadline before *scheduler2* will run next, and therefore sends the reservation to sched_group1, where

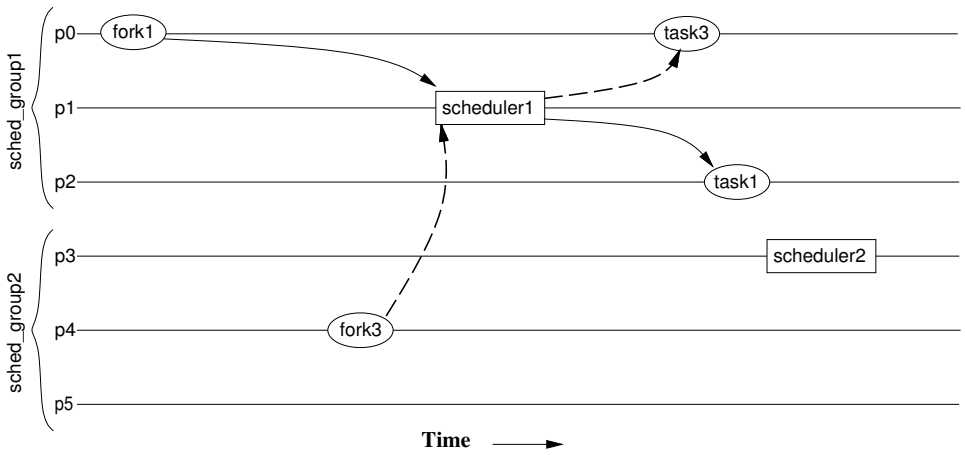

Figure 2: Operation of the Multiprocessor Scheduler

scheduler1 schedules task3 on p0. In addition to knowledge about the frequencies of scheduler execution, processor load information can also be made available to the fork procedure, thereby permitting it to decide where schedulability analysis is most likely to succeed [29].

The operational description of scheduling in Figure 2 permits us to define several terms precisely. First, *scheduling latency* is defined as the total elapsed time from generation of a scheduling request (reservation generation) to insertion of the task into a designated task list. Scheduling latency is affected by the degree of concurrency of scheduling, by the rate of execution of scheduling threads, by the costs of scheduling mechanisms including RLs, SLs, TLs, and reservation entries, and by the schedule of the requesting thread, which makes the final decision concerning task scheduling. Minimum scheduling latency is attained when the requesting thread itself performs schedulability analysis, decision making, and task insertion not using reservations and reservation lists. Measurements of `Rapid` comparing this minimum latency to expected latency values will be provided with the final version of this paper. Tradeoffs in scheduling latency vs. scheduling quality are apparent when newly arriving tasks are rejected simply because their local scheduling group is full whereas other scheduling groups have resources that may be utilized by the task.

Scheduling throughput is defined as the number of tasks scheduled over time by some configuration of the `Rapid` scheduler. Throughput is directly related to scheduling latency when sets of tasks must be co-scheduled, but increased throughput due to concurrency in scheduling can result in increased latency for individual tasks due to the additional overheads of reservations and reservation lists and due to the additional delays caused by possible gaps between the execution times of the requesting thread and the various scheduling threads. Such issues may be addressed by proper scheduling of scheduling threads, such as the staggered schedule suggested by Zhou[30]. In order to evaluate scheduling overheads and latencies without considering this issue, we also define *scheduling effort* as the total amount of execution time spent on performing scheduling actions on behalf of a single task. Clearly, best fit policies will differ significantly in scheduling effort from any fit, and scheduling effort when the requesting task performs all scheduling actions will differ due to reservation overheads from the total effort made by asynchronously executed scheduling

threads on behalf of this task.

4 Rapid Performance

The purpose of this section is to evaluate the scheduling technology offered by Rapid. We focus on the novel contributions of this work, which are:

- *Concurrency* in scheduling – what are the effects of concurrency on both the quality and performance of scheduling, where performance is measured as scheduling latency and as scheduling effort for individual and sets of tasks?
- *Implementation* issues for on-line scheduling – to evaluate the effects of decoupling task generation, from schedulability analysis, from decision-making, and to understand issues specific to large-scale parallel machine, such as appropriate implementations of reservations and reservation lists which determine the manner in which tasks are shared by multiple schedulers.
- *Configuration* of schedulers – what are the costs vs. benefits of being able to configure schedulers leading to different sizes of scheduling groups and different structures of reservation lists?

4.1 Performance of Rapid Mechanisms

We first evaluate basic performance properties of the Rapid scheduler. All such measurements are obtained from runs with simulated workloads on a Kendall Square Supercomputer (KSR-1), which has a 0.05μsec clock cycle time and memory access times ranging from 2 to 600 processor cycles depending on the required caching actions.

The costs of request generation (ie., reservation generation) and selection from reservation lists are small, roughly 30 μseconds total. Comparatively higher are the costs (1) of reservation entry (in deadline order) into the reservation list, which depends on list size and (2) the costs of schedulability analysis. (1) is not interesting to evaluate since a number of obvious optimizations not yet applied to Rapid can be performed, including the use of hashing, of calendar queues, and of access trees. (2) requires approximately 180 μseconds for a single processor's slot list. Additional details on some of these costs will appear in an extended version of this paper, including the costs of reservation generation, minimal decision making consisting of diagnosing success or failure, and insertion into the local task list. Significant variations in execution times only exist for list (RL, SL, and TL) manipulation, with total execution time dominated by SL manipulation. Details on the effects of task laxities, execution times, and system load on SL execution times are not reported below; they are described in a previous publication[30]. For purposes of this paper, it is sufficient to state that SL execution time strictly depends on the number of slots in the list, which tends to remain constant and/or grow smaller with increasing system load due to the merging of adjacent slots.

Conclusions from these results are straightforward. First, additional scheduling costs due to the use of reservations is small compared to the cost of schedulability analysis, prompting us not to consider implementation of an optimized path for local scheduling in which reservations are not used. Second, parallelization of on-line scheduling should

be most effective when applied to schedulability analysis, especially when each task must be analyzed for multiple processors or when sets of tasks must be evaluated, as is the case for the co-scheduling of groups of tasks. Third, even assuming fairly moderate request rates for schedulers, a single scheduler cannot handle more than a few processors, again due to the cost of schedulability analysis. This indicates that concurrency in scheduling is essential for larger scale parallel system or that hardware support is required for scheduling as described in [19].

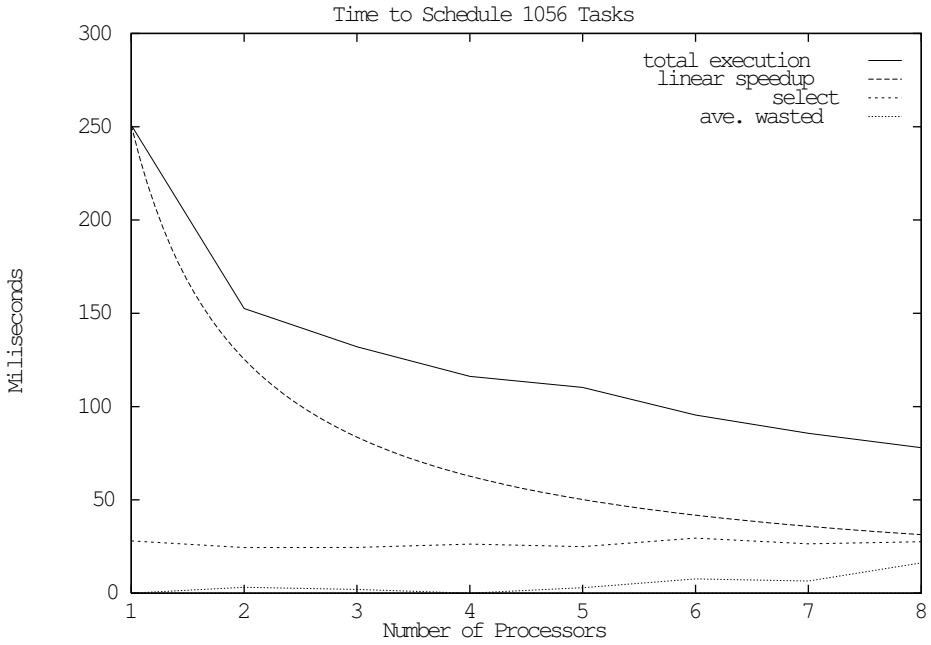

Figure 3: Throughput of feasible schedule with a remote RL

4.2 Effects of Concurrency on Scheduler Performance

Scheduling throughput. Performance improvements due to the use of concurrency are shown in Figure 3, where *scheduling throughput* is demonstrated as the total time required to schedule a remote fixed set of feasible tasks already available when schedulability analysis is initiated. Several specific times are reported. First, the time required for *request selection* is relatively flat, which indicates that contention for requests is minimal. *Average wasted time* is the time spent analyzing a reservation that is ultimately declined. This happens because of programmed permitted asynchrony and duplication in request selection. A strong restriction (mutex lock) on reservation list access would preclude this possibility, but significantly increases request selection time.

This graph also shows that speedup declines fairly quickly, indicating a limitation on the number of schedulers that can reasonably share a single request list. Detailed measurements of earlier implementations of the reservation list exposed several implementation issues not reported in detail in this paper.

Scheduling quality. One possible undesirable effect of concurrent scheduling is a decrease in schedule quality. `Rapid` guarantees that reservations will be examined in their listed order (for which a least laxity ordering has been

N	Schedulable Task Sets	
	Zhou	Rapid
100	100	100
200	100	100
300	100	100
400	100	100
500	100	100
600	100	100
700	99	100
800	90	95
900	72	81
1000	32	46

Table 1: Scheduling quality comparison with Zhou

shown optimal by Zhou for off-line multiprocessor scheduling). However, requests are not guaranteed to be scheduled in the order in which they appear in the list due to possible differences in execution speeds of scheduling threads.

Table 4.2 compares `Rapid's` schedule quality to Zhou's slot list scheduler. Both schedulers used the same set of input tasks with exponentially distributed start times, run times and deadlines. The results are reported as the number of task sets out of 100 where all tasks are schedulable. These results show that the concurrent scheduling in `Rapid` does not adversely impact schedule quality. Improvements in concurrent vs. non-concurrent scheduling for heavily loaded systems are due to

Hierarchical scheduling. Figure 4 demonstrates the importance of hierarchal scheduling. This graph shows the total time required to schedule 1056 randomly (exponential distribution) generated tasks from a remote request list. As seen in this graph, total execution time of schedulability analysis is dominated by the amount of time spent analyzing requests that ultimately fail, except in the rightmost part of the graph where the task rejection rate falls to zero. The failed time vastly exceeds the amount of time spent analyzing request that ultimately succeed (useful). Therefore, early rejection of tasks that obviously cannot be scheduled via primary scheduling can significantly reduce scheduling overhead and potentially decrease the scheduling latency if the application uses an adapt and reschedule strategy like `slow_and_reschedule` described previously.

5 Conclusions and Future Work

Conclusions. The use of concurrency during schedulability analysis is shown to increase the performance of on-line real-time scheduling. Such increases outweigh the costs of additional mechanisms required for asynchronous task generation, schedulability analysis, and decision-making. Furthermore, increased performance is shown not to

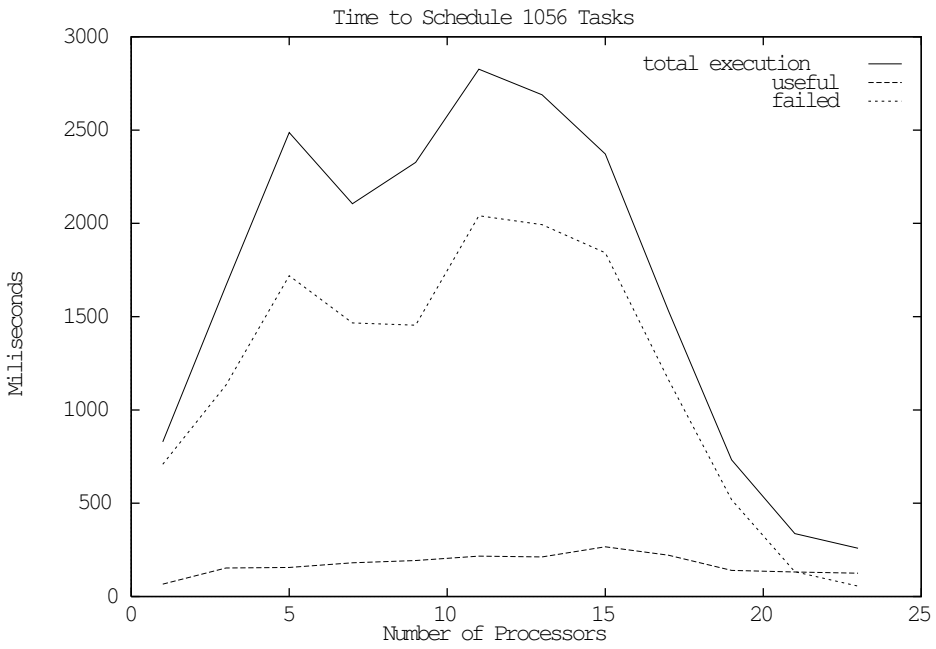

Figure 4: Throughput for a random (infeasible) schedule

degrade the quality of on-line multiprocessor scheduling significantly compared to our past results attained with synchronous scheduling methods[30]. The primary `Rapid` concept and mechanism supporting such asynchrony is the notion of *reservation*, which defines an application interface containing scheduling information to end users and permits concurrent scheduling components to share information concerning single or groups of tasks being scheduled, thereby also defining interfaces between those components.

`Rapid`'s implementation directly supports its extension to large-scale parallel machines, because its data structures are distributed across the different nodes of a parallel machine, accessed locally to increase locality of access, and because `Rapid` itself can be configured to vary the number of processors for which individual schedulers are responsible, the number of processors to be considered for each task being scheduled, and the scheduling algorithm used for multiprocessor task scheduling (e.g., best fit or any fit). The concept supporting such configurability is the *scheduling group*, defined as a number of processors served by a single scheduler, but each retaining individual data structures for threads scheduling and schedulability analysis. Tasks are scheduled within their groups, whenever possible, and are shared between multiple groups with explicit inter-group communications using a reservation list. Reservation list fragmentation vs. the use of a central list can result in significant execution time reductions due to decreases in caching and/or remote memory access on the underlying parallel machine, as shown in [8].

While the results presented in this paper address the efficient scheduling of single or sets of tasks, `Rapid` already contains support for groups of tasks that must be co-scheduled with 'all or none' semantics, or must be partially scheduled separating essential from non-essential tasks, etc. It is our intent to experiment with such scheduling, in part to demonstrate the performance gains possible from the use of parallelism for scheduling tasks related by

dependency relationships. In addition, it should be apparent that the co-scheduling of several tasks on different processors is equivalent to the scheduling of a single task on one processor jointly with the allocation of one or multiple schedulable resources required by this task on schedulable 'processors' interpreted as resources. `Rapid` can address such dynamic task and resource scheduling with minor changes to its current implementation.

Not addressed extensively by `Rapid` is how to deal with failures in task scheduling, in task and resource scheduling, etc. Instead, `Rapid` returns failure and timing information to the task generator, then expecting the user to implement recovery actions, submit alternate tasks for schedulability analysis, etc. In addition, `Rapid` permits users to specify failure functions executed by scheduling threads in reaction to scheduling failures. Our motivation for not including additional functionality to address exception handling is the conviction that such functionality ought to reside at higher levels of the real-time operating system, as described in [10], or at user level where application semantics may be used in making recovery decisions. For example, in the robot navigation code described in Section 2, planning is performed continuously during robot movement so that partially generated alternate plans are continuously available. However, it is up to the application to decide on the amounts of re-planning required upon obstacle detection and then submit for re-scheduling the previously scheduled planning tasks. `Rapid` offers no higher-level support for ease of implementation of such forward recovery. Note that the use of pre-scheduling or even imprecise computation[14] may be indicated in this context system loads are sufficiently high to result in a high probability of failure for dynamic re-scheduling.

Future research. The immediate next steps in this research concern experimentation with task groups, where different tasks in the same group must be co-scheduled. Additional performance and quality evaluation of on-line scheduling will be performed, as well. More importantly, we will generalize the current `Rapid` scheduler to provide: (1) application interfaces to support the robotics applications described in this paper and additional real-time applications, resulting in an alternative to the real-time threads implementation described in [22], and (2) increased configurability for inclusion of resource scheduling algorithms and of different models of timing constraints, and for experimentation with scaling `Rapid` to larger parallel machines, where alternative scheduling groups and communication structures among those groups must be considered[8]. In essence, `Rapid` will be the basis for a flexible framework for implementation of schedulers for embedded systems. Last, we will experiment with task groups that require the co-scheduling of related tasks or that require the allocation of both CPU and other resources for task execution.

In the course of developing our scheduler, we have recognized the need to develop several new scheduling policies beyond the standard best-fit, any-fit and worst-fit algorithms.

For multi-threaded tasks where the application programmer supplies cooperation controls, the scheduling policy must distribute the task's total runtime among a set of processors. Two reasonable policies to accomplish this would be a greedy algorithm and an even distribution algorithm. A greedy algorithm would assign each processor as much of the task as possible between the task's start time and deadline. An even distribution algorithm would attempt to assign the same amount of time to each of the processors.

Likewise, for a set of related tasks that must be scheduled with all-or-none semantics, there could be greedy

and even distribution policies where each processor is assigned as many tasks as possible (greedy) and where each processor is assigned the same amount of total runtime from the group (even distribution).

There is an extension to strict alternate scheduling where only one task of a set must be scheduled where scheduling some subset of the group is considered successful. Each individual task would have an associated scheduling success quantity. If the total of these quantities for the successfully scheduled tasks is greater than the application defined criteria, then scheduling succeeds. Otherwise, scheduling fails and none of the set of tasks is scheduled.

We have also recognized that efficient multiprocessor scheduler architectures are dependent on the underlying hardware characteristics, particularly the local/remote memory access ratio. While the current implementation is reasonably flexible, there are several areas that need further study.

In the three phases of scheduling (offer generation, schedulability analysis, and offer resolution), offer resolution is currently ad hoc. In some scheduling policies, resolution is implicit (e.g. any-fit). In other policies (e.g. best-fit) resolution is explicit following some termination condition. In the case of real-time systems, schedulability analysis termination could be either all processors/tasks analyzed or the arrival of the task's start time. Offer resolution should be formally separated from schedulability analysis to provide greater parallelism, predictability, and more complete schedulability analysis.

We also recognize that it may be useful to have schedulers exchange offers between themselves rather than relying on the fork mechanism sending the offer to the appropriate scheduler. Schedulers could be connected in rings or trees or other logical structures based on the application characteristics and the hardware characteristics.

The time cost of sharing offers inherent in the hardware and the acceptable scheduling latency defined by the application, will determine the structure of the scheduling groups. At one extreme, very fast inter-processor communications and very short latency requirements will result in a single scheduling group which includes all the processors of the system. At the other extreme, a ring of scheduling groups consisting of a single processor each, results from high communications costs and long scheduling latencies.

The connections between scheduling groups need not be uniform or static. In an application where task generation is not uniform in time or across tasks, scheduling groups which generate more than the average number of tasks should have more connections to other scheduling groups than groups which generate fewer tasks. An implementation of this principle might have scheduling groups with long offer queues adding connections to scheduling groups with short offer queues and groups with short offer queues deleting connections to groups with long offer queues. The assumption here is that the task rejection rate will be relatively small, so that the offer queue length is related to the future processor load.

Acknowledgements. Tucker Balch is contributing the robot navigation and planning code used in this research. Hongyi Zhou has provided some assistance in comparing the quality of `Rapid` scheduling to her previous results published in [30].

References

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. Technical report, Department of Computer Science and Engineering, University of Washington, TR 90-04-02, April 1990.

[2] R.C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, August 1989.

[3] R.C. Arkin. The impact of cybernetics on the design of a mobile robot system: A case study. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1245–1257, Nov/Dec 1990.

[4] T. Balch, H. Forbes, and K. Schwan. Dynamic scheduling for mobile robots. In *6th EuroMicro Workshop*, June 1994. to appear.

[5] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–147, May 1991.

[6] Tomas E Bihari, Thomas M Walliser, and Mark R Patterson. Controlling the adaptive suspension vehicle. *IEEE Computer*, pages 59–64, June 1989.

[7] Ben Blake. *A Fast, Effective Scheduling Framework for Parallel Computing Systems.* PhD thesis, Department of Computer and Information Science, Ohio State University, 1990. C-128-B.

[8] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (dsa) on large-scale multiprocessors. In *Proc. of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 227–246. USENIX, September 1993.

[9] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[10] Ahmed Gheith and Karsten Schwan. Chaos-arc - kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[11] Ahmed M. Gheith. *Support for Multi-Weight Objects, Invocations and Atomicity in Real-Time Systems.* PhD thesis, Georgia Institute of Technology, Atlanta, GA 30332-0280, December 1990.

[12] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106–125, July 1989. Also availble as Philips Technical Note TN-89-006.

[13] Dilip D. Kandlur, Daniel L. Kiskis, and Kang G. Shin. HARTOS: A distributed real-time operating system. In *Operating Systems Review of ACM SIGOPS*, pages 72–89. ACM Press, July 1989.

[14] Kwei-Jay Lin, Swaminathan Natarajan, and Jane W S Liu. Imprecise results: Utilizing partial computation in real-time systems. In *Proceedings Real-Time Systems Symposium*, pages 210–217, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1903, 1987. IEEE Computer Society, IEEE Computer Society Press.

[15] Jim McDonald and Karsten Schwan. Ada dynamic load control mechanisms for distributed embedded battle management systems. In *First Workshop on Real-time Applications, New York*, pages 156–160. IEEE, May 1993.

[16] Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace, and Charles Weems. The spring scheduling co-processor: Design, use and performance. In *Proceedings of the Real-Time Systems Symposium*, pages 106–111. IEEE Computer Society Press, December 1993.

[17] David W. Payton and Thomas E. Bihari. Intelligent real-time control of robotic vehicles. *Communications of the ACM*, 34(8):48–63, August 1991.

[18] K. Ramamritham, J. Stankovic, and Wl Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, pages 1,110–1,123, August 1989.

[19] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 1(2), pages 184–194, April 1990.

[20] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[21] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A c thread library for multiprocessors. Technical Report TR-91/02, Georgia Institute of Technology, Atlanta, GA 30332-0280, January 1991.

[22] Karsten Schwan, Ahmed Gheith, and Hongyi Zhou. Dynamic scheduling for hard real-time systems: Toward real-time threads. In *Joint IEEE Workshop on Real-Time Operating Systems and Software and IFAC Workshop on Real-Time Programming, Atlanta, GA*, pages 13–21. IEEE, May 1991. Also in IEEE Real-Time Systems Newsletter, Vol. 7, No. 4, Fall 1991, pp. 14-22.

[23] 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, Aug. 1992.

[24] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[25] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[26] Anthony Stentz. *The NAVLAB System for Mobile Robot Navigation.* PhD thesis, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1990.

[27] Anthony Stentz. Optimal and efficient path planning for partially-known iguanas. In *International Conference on Robotics and Automation.* IEEE, May 1994. to appear.

[28] R. W. Wisniewski and C. M. Brown. Ephor, a run-time environment for parallel intelligent applications. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, pages 51–60, April 1993. Newport Beach, CA.

[29] Hongyi Zhou. *Task Scheduling and Synchronization for Multiprocessor Real-Time Systems.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, April 1992.

[30] Hongyi Zhou, Karsten Schwan, and Ian F Akyildiz. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. In *Real-Time Systems Symposium.* IEEE, IEEE, 1992. Also available as GIT-CC-91/40 from Ga. Tech.