Efficient and Predictable Process Scheduling

M.V. Panduranga Rao, K.C. Shet[1] and K. Roopa[2]

National Institute of Technology Karnataka, Mangalore
[1]Department of Computer Engineering, National Institute of Technology, Surathkal, Karnataka
[2]Mphasis Limited, Bangalore
E-mail : raomvp@yahoo.com ; kcshet@yahoo.co.uk ; roopa.sindhe@gmail.com

ABSTRACT

Real-time applications such as multimedia audio and video are increasingly populating the workstation desktop. To suppor tthe execution of these applications in conjunction with traditional non-realtime applications, we have created a parametric scheduler for multimedia and Real-Time applications. Workstations and personal computers are increasingly being used for applications with real-time characteristics such as speech understanding and synthesis, media computations and I/O, and animation, often concurrently executed with traditional non-real-time workloads. This paper presents a system that can schedule multiple independent activities so that:

- *Activities can obtain minimum guaranteed execution rates with application-specified reservation granularities via CPU Reservations.*
- *CPU Reservations, which are of the form "reserve X units of time out of every Y units", provide not just an average case execution rate of X/Y over long periods of time, but the stronger guarantee that from any instant of time, by Y time units later, the activity will have executed for at least X time units.*

Keywords: *scheduler, dispatcher, RTOS, deadline, graph, rms, edf, preemption, DDS, CPU Reservations.*

1. INTRODUCTION

1.1 Scheduler

The scheduler's job is to allocate shared resources to tasks. Those resources could be CPU, peripheral, etc. The scheduler is driven through a clock tick. Each task is assigned a period (which assigns the frequency of execution) and a due time (which assigns the phase of execution).

1.2 Deadline

The deadline is used to temporally limit a task's allocation of resources. A task which exceeds its allotted time will be suspended by the scheduler regardless of whether or not it has completed. This frees up a resource for other tasks and prevents an errant task from seizing shared resources.

1.3 Deadline-Driven Scheduler (DDS)

DDS s use the deadline of a task to make decisions as to which task has to be dispatched next. At the so called "task release" time (the time this task became available for scheduling), DDS will re-order the ready task queue according to increasing deadline order. The ready queue consists of ready but unfinished tasks. The dispatcher then (re)starts the ready task with the earliest deadline, and hence the earliest deadline-first scheduling.

The above case does not consider the possibility that tasks may miss their deadlines. If all tasks have hard-deadlines and need to be "guaranteed", the new task will not be entered in the ready queue, unless there is enough time in the system to accomplish this task and all other previously accepted tasks. If there is not enough time, this task may be rejected.

It is not necessary that the task is a periodic one (as it is implied below), although periodic tasks can be dispatched by a deadline driven scheduler.

1.4 At runtime

- Select the task with the highest criticalness (the highest order bits).
- If two or more tasks have the same criticalness then select the task with the minimum laxity (deadline-execution).
- If two or more tasks have the same criticalness and laxity then first come first serve.
- This algorithm gives us the best tradeoff between flexibility and determinism.

A *Scheduling System* consists of two conceptual parts, (*a*) a *Scheduler* and (*b*) a *Dispatcher*. Schedulers develop the task schedules, determining the exact points in time that resources are allocated to tasks. Dispatchers are the mechanisms that implement schedules. Schedules must satisfy constraints imposed by the operational objectives of the system.

Schedules can be priority-based or time-driven. Priority-based schedules are more flexible and powerful. They fall broadly into two priority assignment methods: (*a*) the fixed and (*b*) the dynamic priority assignment methods [14].

Under **fixed priority** (FP), static priorities are assigned to tasks, *off-line*. During run-time, the dispatcher uses these priorities to determine which task is going to execute when. Popular FP rules include the *Rate-Monotonic* (RM) and the *Deadline-Monotonic* (DM). The RM and DM rules have been shown to be *optimal* under the following conditions. RM can be used when, $T_i = D_i$, that is when the periods and the relative deadlines are identical for each periodic task *i* and all periodic tasks are initiated at the same time *e.g.*, at $t = 0$. RM assigns fixed priorities to periodic tasks so that tasks with smaller periods get a higher priority. DM is used when for some periodic tasks *j*, $T_j < D_j$, that is, when there are tasks with relative deadlines that are earlier than the tasks' periods. DM assigns priorities so that tasks with shorter deadlines receive higher priorities. If not all tasks are initiated at $t = 0$, there is no polynomial algorithm to find the priority assignment that produces a *feasible* schedule, and determine if the tasks set is schedulable. It has been shown that, under a *given* priority assignment we need to check ("simulate") the entire schedule in $[0, r+H)$, where $r := \max_j\{r_j\}$, r_j is the time since $t = 0$ where task *j* is initiated for the first time, and *H* is the Least-Common Multiple (LCM) of all task periods. We note that in the worst case $H = T_1 \times T_2 \times \cdots \times T_n$, if the periods are relatively prime. This can effectively lead to very long hyper-periods.

Under a dynamic priority rule, priorities are assigned to tasks dynamically upon their arrival at the system. In general purpose schedules priorities are based on some task attribute, such as the release time, the deadline or the execution requirement. In RTS the Earliest Due Date (EDD) or the Least Laxity First (LLF) rules are used, where the task with the earliest deadline, or the task with the least laxity, respectively, receive higher priority [6] [8].

2. DEADLINES

The best-known example of a time constraint is a deadline, but it is a simple one and there are many others (as we will discuss). The use of deadlines in real-time computing is a relatively recent small fraction of the overall theory and practice of deadline-based resource management in various fields (notably logistical fields).

In particular, the real-time computing community historically focuses primarily on hard deadlines (e.g., [ ]). The traditional model of an action having a *hard deadline* at time $\mathbf{t_d}$ is simply that the action's completion either meets or misses its deadline, as illustrated in Fig. 1.

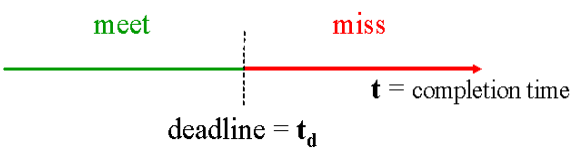

Fig. 1: Traditional Hard Deadline

The semantics of a deadline — i.e., the specific way in which system timeliness depends on whether any particular deadline is met, such as whether a miss constitutes a failure — is not part of the definition of a deadline, contrary to popular misconception in the real-time computing community [11] [19] [13].

When the term "deadline" is used without the qualifier "hard," it refers to the general case of a deadline — a soft deadline, of which a hard deadline is a special case: the action is either more or less timely, depending on what its completion time is with respect to its deadline $\mathbf{t_d}$ [2].

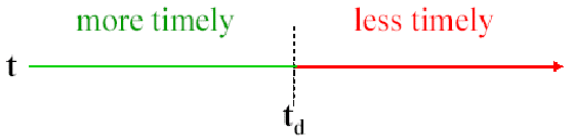

Fig. 2: Traditional Deadline

"More or less" timely with respect to a deadline is always measured in terms of "lateness" and its derivative cases "tardiness" and (to a lesser extent) "earliness:" as illustrated in Fig. 3.

- *lateness* = completion time - deadline
- *tardiness* = max[0,lateness]
- *earliness* = max[-lateness,0]

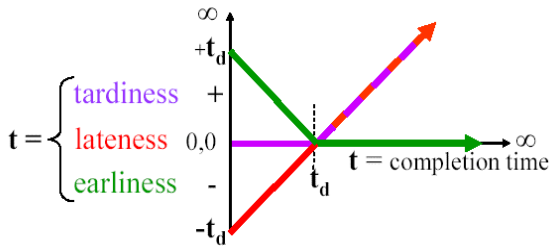

Fig. 3: Deadline Timeliness Metrics

The best (most timely) case with respect to a deadline is that the action completes in zero time, so

- lateness = completion time - deadline = 0 - $\mathbf{t_d}$ = $\mathbf{-t_d}$
- tardiness = max[0,lateness] = max[0,$\mathbf{t_d}$] = 0
- earliness = max[-lateness,0] = max[$\mathbf{t_d}$,0] = $\mathbf{t_d}$

The case where the action completes at its deadline results in

- lateness = completion time - deadline = $\mathbf{t_d}$ - $\mathbf{t_d}$ = 0
- tardiness = max[0,lateness] = max[0,0] = 0
- earliness = max[-lateness,0] = max[0,0] = 0

An example case where the action completes at twice its deadline results in

- lateness = completion time - deadline = $\mathbf{2*t_d}$ - $\mathbf{t_d}$ = $\mathbf{t_d}$
- tardiness = max[0,lateness] = max[0,$\mathbf{t_d}$] = $\mathbf{t_d}$
- earliness = max[-lateness,0] = max[$\mathbf{-t_d}$,0] = 0

On the next page, we show that deadlines are a special, limited, case of a more general and expressive model of time constraints: time/utility functions.

3. SCHEDULER IMPLEMENTATION

Three major aspects of the scheduler implementation are discussed in this section:

- **Precomputed Scheduling Graph:** A *scheduling graph* is precomputed from the set of CPU reservations, preallocating sufficient time to satisfy all reservations on an ongoing basis.
- **Time Interval Assignment:** Specific *time intervals* are set aside within the scheduling graph for the execution of feasible time constraints.
- **EDF Constraint Execution:** Feasible constraints are executed in Earliest Deadline First order.

3.1 Precomputed Scheduling Graph

The fundamental basis of this scheduling work is the ability to precompute a repeating schedule such that all accepted CPU reservations can be honored on a continuing basis and accurate feasibility analysis of time constraints can be performed [11]. Furthermore, this schedule may be represented in a data structure

that may be used at run-time to decide, in time bounded by a constant, which activity to run next. We currently represent this precomputed schedule as a binary tree, although more generally it could be a directed graph [5].

Fig. 4. Shows a scheduling graph for six activities with the following reservations:

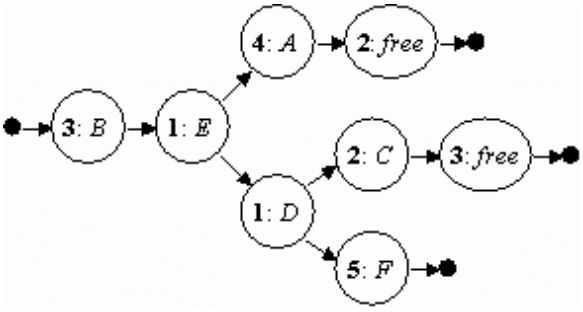

Fig. 4: Scheduling graph with a base period of 10ms

Table 1:

Activity	*A*	*B*	*C*	*D*	*E*	*F*
Amount	4ms	3ms	2ms	1ms	1ms	5ms
Period	20ms	10ms	40ms	20ms	10ms	40ms

Each node in the graph represents a periodic interval of time that is either dedicated to the execution of a particular activity or is *free*. For instance, the leftmost node dedicates 3ms for the execution of activity *B*.

Each left-to-right path through the graph is the same length, in this case 10ms. This length is the *base period* of the scheduling graph and corresponds to the minimum active reservation period [21].

The scheduler repeatedly traverses the graph from left to right, alternating choices each time a branching point is reached. When the right ends of the graph are reached, traversal resumes again at the left. In this example, the schedule execution order is:

(*B*, 3ms), (*E*, 1ms), (*A* , 4ms), (*free*, 2ms), (*B*, 3ms), (*E*, 1ms), (*D*, 1ms), (*C*, 2ms), (*free*, 3ms), (*B*, 3ms), (*E*, 1ms), (*A*, 4ms), (*free*, 2ms), (*B*, 3ms), (*E*, 1ms), (*D*, 1ms), (*F*, 5ms).

After this the schedule repeats.

The execution times associated with schedule graph nodes are periodic and fixed during the lifetime of the graph; they do not drift. For instance, if activity *D* is first scheduled to run during the time interval [*T*, *T*+1ms], it will next run during [*T*+20ms, *T*+21ms], then [*T*+40ms, *T*+41ms], etc. Of course, when an interrupt occurs, whatever time it takes is unavailable to the node intervals in which it executes, causing them to receive less time than planned. However, the effects of such an event are limited to the executing activity and thread, rather than being propagated into the future. Allowing perturbations to affect the future execution of the scheduling plan would have disastrous effects to the satisfiability of already granted time constraints.

Each node following a branching point in the graph is scheduled only half as often as those preceding it. For instance, activity *A* is only scheduled every 20ms — half as often as activity *E* at 10ms. Likewise, *C* is scheduled every 40ms, half as often as *D*. This makes it possible to schedule reservations with different periods using the same graph, provided that each reservation period is a power-of-two multiple of the base period [1] [4].

Reservations where the period is not such a multiple are scaled and scheduled at the next smaller power-of-two multiple of the base period. For instance, *A* might have originally requested 6ms every 30ms but because the base period of the graph is 10ms, its reservation was actually granted at 4ms per 20ms — the same CPU percentage but at a higher frequency. Applications are told the actual reservation granted, allowing them to iterate and change their reservation request in response if they deem it appropriate.

The scheduling graph must be updated upon CPU reservation changes. Criteria used in graph construction include minimizing the number of context switches between activities and maximizing the time slices. The graph is constructed to allow sufficient time for the context switches between nodes. See Section 4.1 for a more complete description of graph construction [3].

Benefits of the precomputed scheduling graph include:

- CPU reservations are enforced with essentially no additional run-time scheduling overhead. The scheduling decision involves only a small number of pointer indirections. This number is bounded by a constant and is independent of the number of threads, activities, and time constraints.
- Accurate feasibility analysis for time constraints can be performed because it is known in advance when an activity will be executed and when free time intervals will occur.

3.2 Time Interval Assignment

The scheduler analyzes the feasibility of each time constraint when a thread submits it. If the constraint is determined to be feasible, the BeginConstraint() call returns a success status code and the scheduler promises that at least the estimated amount of time has been reserved for the constraint's execution between the specified start time and deadline. If infeasible, a different status code is returned and no time is reserved. Constraint feasibility analysis and reserving time for accepted constraints are both accomplished using *time interval assignment* data structures [19].

Recall that the scheduling graph allows us to know in advance both the time intervals during which each activity will be run and the time intervals that are not dedicated to any activity. Thus, it is possible to assign specific future time intervals for the execution of the time constraint, both from the activity of the thread requesting the time constraint and from the free intervals. If sufficient time intervals are available over the lifetime of the constraint, then they are marked as assigned to the execution of that constraint. In this case, the constraint is feasible and the requesting thread receives a success status code from the BeginConstraint() call. Otherwise, the constraint is infeasible [17].

Each node in the scheduling graph has a (possibly empty) list of interval assignment records associated with it, ordered by start time. Interval assignments are made by adding references to interval assignment records to scheduling graph nodes covering the interval's time period. These records contain the start and end times of the assigned interval within the node, and a reference to the constraint for which the assignment is being made.

Note that a time constraint will not result in changes to the underlying precomputed scheduling graph, but only in adding interval assignment records to the lists in some of the graph nodes.

As an example, suppose that the time constraints below are issued by threads of the activities listed at the times given, with CPU reservations as in the previous example:

Table 2:

	Activity	**Issue Time**	**Estimate**	**Start Time**	**Deadline**
C_1	*A*	205ms	11ms	230ms	270ms
C_2	*E*	213ms	11ms	215ms	265ms
C_3	*A*	225ms	10ms	225ms	270ms

and the next several scheduling graph nodes dedicated to activities *A* or *E*, or are free are listed in the first two columns below:

Table 3:

Node Interval (ms)	**Activity**	**@ 205ms**	**@ 213ms**
204 - 208	*A*		
208 - 210	*free*		
213 - 214	*E*		
217 - 220	*free*		$C_2(3)$
223 - 224	*E*		$C_2(1)$
224 - 228	*A*		
228 - 230	*free*		$C_2(2)$
233 - 234	*E*		$C_2(1)$
243 - 244	*E*		$C_2(1)$
244 - 248	*A*	$C_1(4)$	$C_1(4)$
248 - 250	*free*	$C_1(2)$	$C_1(2)$
253 - 254	*E*		$C_2(1)$
257 - 260	*free*	$C_1(1)$	$C_1(1), C_2(1)$
263 - 264	*E*		$C_2(1)$
264 - 268	*A*	$C_1(4)$	$C_1(4)$
268 - 270	*free*		

Assuming that no previous interval assignments had been made to those intervals, the feasibility analysis for constraints C_1 and C_2 will succeed, resulting in the interval assignments listed in the last two columns, and the feasibility analysis for C_3 will fail.

The interval assignment procedure first assigns node intervals already dedicated to the requesting activity, assigning free node intervals only if needed. This rule explains why C_1 is assigned 4ms in the interval (264-268ms) and only 1ms from the earlier free interval (257-260ms). Note that the interval (259-260ms) remains available for other assignments.

During this procedure, when a node is visited, all available time intervals are assigned to the constraint up to the required estimate. The procedure stops once the estimate is reached.

C_3 cannot be guaranteed. It could be assigned only the intervals (225-228ms), (259-260ms), and (268-270ms) — just 6ms of the needed 10ms. When the analysis fails, any tentatively assigned intervals are deassigned [20].

4. ADDITIONAL SCHEDULER DETAILS

4.1 Scheduling Graph Computation

The precomputed CPU scheduling graph is the foundation upon which guaranteed CPU reservations, accurate time constraint feasibility analysis, and guaranteed time constraints are built, all while keeping the context switch overhead low and independent of the system load. Special attention is paid to minimizing this overhead, as we are targeting large systems, possibly with hundreds of concurrent activities [2] [7] [18].

As already stated, the precomputed scheduling plan of our prototype is a binary tree; more complex data structures, such as trees with variable branching-factor, could be used. Independent of the selected representation, the precomputed scheduling plan must satisfy a minimal set of requirements: account for context switch overheads, minimize the number of context switches, and distribute free CPU evenly over time.

The context switch time must be accounted for in the scheduling plan in order to have an accurate representation of CPU usage. This is required for a precise feasibility analysis of time constraints. Likewise, it is also particularly important when there are activities that require very small and frequent execution intervals (e.g., with a period on the order of 1ms).

Unnecessary context switches are avoided by scheduling activities as close as possible to their desired periodicity and with as few execution intervals per period as possible (preferably one). Also, to avoid wasting CPU, all the execution intervals in the scheduling plan are required to be larger than a certain minimum [15].

Evenly distributing the free CPU intervals over time increases the chances for time constraints to be able to use these intervals (if needed), irrespective of their start times or deadlines. Likewise, given a relatively uniform distribution of the free CPU, a legacy scheduler (see Section 3.5) can be assigned a "uniformly slower" CPU to manage, providing a uniform execution rate for timeshared activities (i.e., those with no reservations) [16].

The goals of minimizing the number of context switches and distributing the free CPU uniformly in the general case are very difficult to achieve. Consider the problem of assigning a set of activities with periods T and 2T. This will result in a scheduling graph with nodes on one branch of period T and on two branches of period 2T. The general problem of evenly distributing the free CPU between the two 2T-branches is NP-hard.

To avoid this complexity, our algorithm doesn't attempt to compute the best prescheduling plan but to efficiently compute a plan that is "good enough". For instance, we try to incrementally modify the current scheduling plan whenever the new reservation has a period no smaller than the base period of the current plan. If unsuccessful, a new scheduling graph is computed.

The computation of a new scheduling plan uses heuristic search over a quantized representation of the CPU resource. Briefly, the CPU resource is represented as a complete binary tree (called the *availability tree*) of depth $floor(\ log_2\ (maxY/minY)\ )$, where $maxY$ and $minY$ are the maximum and minimum periods among the reservations being considered. Each node of the availability tree represents a recurring execution interval with a period determined

by the node's depth in the tree. This period is ($minY \times 2^{depth}$), where the root has depth zero.

Each potential branch (i.e., sequence of nodes with same period) in a scheduling graph corresponds to exactly one node with the same period in the availability tree. Assigning a reservation to a branch in the scheduling graph is equivalent to finding an acceptable node in the availability tree.

Each node of the availability tree is labeled with the amount of time currently available for assignment within the execution interval it represents. Initially, this amount is the same for all nodes and equal to *minY*. An algorithm invariant is that the label of each availability tree node is always the minimum of the labels of its two children. The labels of sibling nodes are independent.

Reservation assignments are made in decreasing order of percentages. This heuristic improves the performance of the algorithm. Backtracking is triggered whenever a reservation can not be placed given the current tentative assignments. Once all reservations are assigned to branches, the size and position of the all scheduling graph nodes can be determined.

Many such graph construction algorithms are possible. While the simple one outlined above is sufficient for our prototype to demonstrate the benefits of using a precomputed scheduling plan, exploring the space of practical graph construction algorithms is one possible area of future research.

The time spent computing a new plan is charged against the requesting activity and does not interfere with the execution of other activities. In our current implementation, at any time, there can be at most one active computation of a new scheduling plan.

4.2 Next Thread Selection

Upon a timer interrupt or whenever the current thread blocks, the next thread to run is selected using the following rules:

- If the time remaining in the current node is below a minimum threshold, select the next node.
- If the node has an active interval assignment, execute the constraint with the earliest deadline.
- Else if the node is reserved for an activity, select a runnable thread of the activity, if any, choosing in order: (1) threads with pending denied constraints, (2) threads with late constraints, (3) round-robin among runnable threads.
- Else if a briefly blocked activity has become runnable, chose a thread within it to run in the manner of the previous step.
- Else use the CPU interval for the round-robin queue of activities, choosing a thread as previously described.

The following steps determine the actual context switch overhead:

- Update the state of the current thread, activity, and constraint (if any) (O(1)). The constraint update may result in removing it from the EDF list, which also takes O(1) time.
- Determine the next execution interval. This may require moving to the next node. Unless unexpected system events cause execution to fall behind schedule and multiple nodes have to be traversed (and updated), this step is also O(1).
- Select a thread to run on behalf of the current constraint when the node has an active interval assignment. This step may require a traversal of the constraint inheritance list. A path compression algorithm often reduces this traversal to no more than two steps, i.e. O(1). Selecting the current constraint may require traversal of the EDF list until a constraint with a runnable thread is found. This case should be rare, however, since blocking within a constraint "voids its warranty", making its deadline impossible to guarantee.
- Select a thread to run on behalf of an activity (always O(1)).

In summary, the scheduling decision is always an O(1) operation except in the cases where the thread that would normally have been chosen has blocked in a constraint — violating a precondition needed for guarantees to hold.

4.3 Implementation Parameters

Some parameters of our current implementation are:

- Expected context switch overhead. Actual reserved time intervals are extended by this amount to allow time to switch between activities.
- Minimum execution interval (ten times the expected context switch overhead) — no CPU reservation shorter than this is accepted. Used to ensure that the processor has some time to do work other than just switch between activities.
- Maximum reservation round-up (three times the expected context switch overhead) — maximum length of time by which an interval may be extended beyond the amount requested when making a reservation. Small extensions are permitted to reduce fragmentation of the scheduling graph.
- Maximum reservation period (1sec) — provides for a limit on the depth of the scheduling graph.
- Initial reservations for kernel activities (main kernel activity 30ms/300ms (10%), helper activity 15ms/300ms (5%)).

Any particular set of parameter choices obviously imposes some limits on the total number of concurrent reservations that can be accommodated. The actual number of independent reservations that can be accommodated is, of course, highly dependent upon the particular reservations requested. However, it should be understood these choices do *not* place a limit on the number of concurrent activities or threads in the system — just on those with independent CPU reservations.

5. RESULTS

5.1 Low-level Performance Measurements

Determining a good "expected context switch" value to use as spacing between scheduled time intervals is a hard problem, due to variations caused by cache effects. While we presently use the minimum context switch value (20µs on the PC, 50µs on the set-top box) when building the scheduling graph, a "better" value might be something closer to the median or average.

Another relevant performance measure is the time that it takes to switch threads when one thread blocks on a mutex held by another. As previously discussed, the time to establish a CPU reservation may depend greatly upon both the new reservation parameters and the existing reservations. That having been said, a number of simple relevant measurements can be reported.

The time to do an incremental CPU reservation in which only a single graph node is added is 150µs, out of which 19µs are spent modifying the graph. The remainder is the relatively constant overhead associated with the RPC to the kernel and acquiring the appropriate references and locks. Releasing a reservation can always be accomplished in an essentially constant time of 98µs, out of which 11µs are spent modifying the graph.

Fig. 5. Graph, the times to make an intentionally complex cumulative sequence of CPU reservations. Times shown represent only the time to actually modify the scheduling graph and do not include the essentially constant kernel overheads previously described. All requests reserve 400µs, but at varying periods. The sequence of periods is a pattern, which begins [9]: 1s, 1s, 500ms, 1s, 500ms, 250ms, 1s, 500ms, 250ms, 125ms, etc.

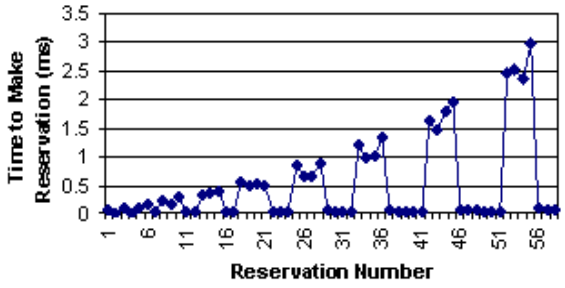

Fig. 5: Scheduling graph construction times.

5.2 Time Constraint Results

Fig. 6. is the graph of the average-case execution time of randomly occurring sporadic tasks, each of which needs 50ms of CPU time to execute. Such tasks may occur in response to user input, such as mouse clicks

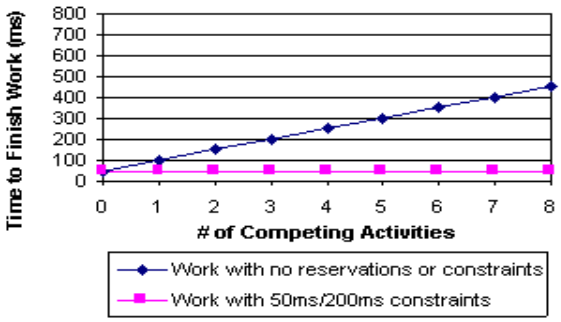

Fig. 6: Execution of 50ms of sporadic work with and without constraints, no reservations.

[10] [12], in response to which a window might need to be redrawn. Response time between the user input and completion of the task plays an important role in the user's perception of the responsiveness of the system. Below 100ms, most people perceive the response as instantaneous. In these experiments, our goal was to do the 50ms of work within 200ms of the randomly occurring aperiodic events (simulating user input) — a just noticeable but acceptable response time.

6. FURTHER RESEARCH

One possible line of future research is to explore integrating this kind of scheduler into general-purpose commercial operating systems with their own scheduling algorithms and policies. A first approach would be to use the legacy scheduling algorithms to schedule free and unused time intervals, although closer integration may be appropriate under some circumstances. While in principle, this should be easy and yield good results, in practice we expect interesting things would be learned along the way.

7. CONCLUDING REMARKS

This research demonstrates the effectiveness and practicality of using a ***Precomputed Scheduling Graph*** both to implement continuously guaranteed ***CPU Reservations*** with application-defined periods and to implement guaranteed ***Time Constraints*** with accurate *a priori* feasibility analysis. Our results show that one need not sacrifice efficiency to gain the predictability benefits of CPU reservations and time constraints.

Furthermore, CPU reservations and time constraints lend themselves to incremental development of real-time applications. Use of CPU reservations and time constraints can be incrementally added both to existing applications and those under development as needed to ensure local and global timeliness properties of the code.

Our experiences gained from implementing and experimenting with the algorithms described in this paper lead us to the conclusion that there is no sound reason why practical, efficient, real-time services enabling independent real-time applications can not and should not be present in nearly all general-purpose operating systems.

8. ACKNOWLEDGMENT

This research work was supported by continuous support of Don Bosco Institute of Technology, Bangalore. The guidance by second author and helped by researchers around India and other countries through emails and information's collected through various bulletin boards.

REFERENCES

1. Chih-Lin Hu, "On-Demand Real-Time Information Dissemination: A General Approach with Fairness, Productivity and Urgency", *21st International Conference on Advanced Information Networking and Applications*, AINA '07, 2007. Page(s):362 – 369, 21-23 May 2007.
2. C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son and M. Marley, "Performance Specifications and Metrics for Adaptive Real-Time Systems," *IEEE Real-Time Systems Symposium*, Orlando, FL, Dec 2006.
3. Jensen 03a, *A Timeliness Paradigm for Mesosynchronous Real-Time Systems*, E. Douglas Jensen, *9th Embedded and Real-Time Applications and Systems Symposium*, May 2005.
4. Jensen et al. 02a, *Guest Editors*, " Introduction to Special Section on Asynchronous Real-Time Distributed System", E. Douglas Jensen and Binoy Ravindran, *IEEE Transactions on Computers*, August 2005.
5. Clark et al. 04, "Software Organization to Facilitate Dynamic Processor Scheduling", Raymond K. Clark, E. Douglas Jensen, and Nicolas F. Rouquette,

Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems, Jan 2007.

6. L. Gauthier, S. Yoo and A. Jerraya, "Automatic generation and targeting of application-specific operating systems and embedded systems software," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11), pp.1293-1301, November 2005.
7. Lu, C., Stankovic, A., Tao, G. and Son, H.S. "Feedback Control Real-time Scheduling: Framework, Modeling and Algorithms", special *issue of Real-Time Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, Vol. 23, No. 1/2 July / September, pp. 85-126, 2002.
8. A. Wierman, and M. Harchol-Balter. "Classifying scheduling policies with respect to unfairness in an M/GI/1". *In Proceedings of ACM Sigmetrics*, 2003.
9. C. D. Gill, D. L. Levine and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real- Time Syst., vol. 20*, pp. 117-154, 2001.
10. W.T. Chan, T.W. Lam and K.S. Mak, "Online Deadline Scheduling with Bounded Energy Efficiency", *Proceedings of the 4th Annual Conference on Theory and Applications of Models of Computation (TAMC)*, 416-427, 2007.
11. M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. "Implementation of SRPT scheduling in web servers". *ACM Transactions on Computer Systems* 21(2): 207-233, 2003.
12. A. Bar-Noy, R. E. Ladner, and T. Tamir. "Windows scheduling as a restricted version of bin packing". In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 224-233, 2004.
13. I. Rai, G. Urvoy-Keller, and E. Biersack. "Analysis of LAS scheduling for job size distributions with high variance". *In Proceedings of ACM Sigmetrics*, 2003.
14. Lam, T., Ngan, T.J. and TO, K. "Performance Guarantee for EDF under Overload", *In proceedings of the Journal of Algorithms, vol. 52*, pp. 193-206, 2004.
15. I. Rai, G. Urvoy-Keller, M. Vernon, and E. Biersack. "Performance modeling of LAS based scheduling in packet switched networks". In *Proceedings of ACM Sigmetrics- Performance*, 2004.
16. H.L. Chan, W.T. Chan, T.W. Lam, L.K. Lee and K.S. Mak, "Energy Efficient Online Deadline Scheduling", *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 795—804, 2007.
17. D. Dyachuk and R. Deters, "Scheduling of Composite Web Services" in DOA'06: *In proceedings of the OTM Workshops 2006, LNCS 4277*, pp. 19 – 20, 2006.
18. S. Aalto, U. Ayesta, and E. Nyberg-Oksanen. "Two-level processor-sharing scheduling disciplines: Mean delay analysis", In *Proceedings.of ACM SIGMETRICS '04*, pages 97–105, 2004.
19. A. Streit. "Evaluation of an Unfair Decider Mechanism for the Self-Tuning dynP Job Scheduler", In Proceedings of the *13th international Heterogeneous Computing Workshop (HCW) at IPDPS*, pages 108 (book of abstracts, paper only on CD). IEEE Computer Society Press, 2004.
20. M.V. Panduranga Rao, Dr. K.C. Shet, K. Roopa and K.J. Sri Prajna, "Implementation of a simple co-routine based scheduler", In *Knowledge based computing systems & Frontier Technologies NCKBFT, MIT Manipal, Karnataka,INDIA.* 19th & 20th Feb 2007.
21. M.V. Panduranga Rao, Dr. K.C. Shet, R. Balakrishna and K. Roopa, "Development of Scheduler for Real Time and Embedded System Domain", *22nd IEEE International Conference on Advanced Information Networking and Applications - Workshops, WAINA '08, 2008*. FINA 2008, Fourth International Symposium on Frontiers in Networking with Applications. Page(s):1 – 6, 25-28 March 2008, *Gino-wan, Okinawa, JAPAN.*