Context-Aware Real Time Scheduling

Kanishka Gupta Azer Bestavros Ibrahim Matta
Computer Science Department
Boston University
Boston, MA 02215
{kanishka, best, matta}@cs.bu.edu

1. Introduction

Motivation: The main thrust of traditional approaches to real-time scheduling is the satisfaction of timing constraints. As such, most of these approaches pay little attention to the optimization of other characteristics of the resulting schedules. Two schedules which may be equivalent in terms of timing constraint satisfaction might have different qualities with respect to other metrics of importance. As an example, consider a wireless sensor node with a directional and/or adjustable power antenna. Besides meeting the deadlines of all the tasks executing on that node, it is also important that the antenna be scheduled in a way that minimizes antenna redirection, power readjustment, and/or maximizes the period of time the antenna is not in use (allowing it to be turned off, for example). Such considerations would result in potentially large power savings. A similar example along these lines is a moving camera tracking a number of objects in a room. Another example that highlights the importance of schedule "quality" is the impact of schedules on locality of reference (with implications on cache efficiency). Other examples are abound.

As the above examples suggest, many emerging applications suggest that real-time scheduling be "context-aware". In other words, the decision of which job to schedule next should be based on the deadline of the job as well as the context of resources being managed.

Model: Consider a set Γ of N periodic tasks. $\Gamma = \{\tau_i | \ \ i = 1...N\}$ where each task is independent and fully preemptive. A task $\tau_i = (c_i, T_i, D_i)$ is characterized by its worst case execution time c_i, its period T_i and a relative deadline D_i equal to or shorter than the period. Jobs are released at the beginning of the period and are eligible for execution immediately after they are released. Let H denote the hyper-period of Γ. Assuming all tasks are released at time 0, a feasible schedule would repeat itself after time H.

For the purposes of this paper, we use the term "context" to refer to the state of one or more resources in the system. The resources of the system can be in n different states (contexts). Tasks belonging to the same class execute in the same context (e.g., with antenna pointing in the same direction). Define Ω to be the context function i.e $\Omega(\tau_i) = j, \ \ i = 1..N, \ \ j = 1..n$ if task i requires the resources to be in context j during its execution. We assume the task to context mapping of the system to be fixed. Furthermore, each task can be in exactly one context. In other words, the contexts partition the task set into n subsets $\Gamma_j, \ \ j = 1...n$ where $\Gamma_j = \{\tau_i | \ \ \Omega(\tau_i) = j\}$ & $\tau_i \in \Gamma\}$

Consider discrete time domain, where time could be normalized by any practical value, be it the time-slice for a CPU, maximum length packet transmission time for a link *etc.* With a slight abuse of notation, let $\Omega(t)$ denote the context of execution at time t.

Problem Statement: Our aim in this paper is to minimize the number of context switches. If we identify contexts with colors, our problem would be akin to generating a feasible schedule such that the number of color changes in the schedule is minimized Figure 1(a) shows a set of 3 periodic tasks, each belonging to a different contexts. The EDF schedule for this set produces 8 context switches in the hyper-period.The challenge lies in meeting the conflicting requirements of reducing changes in contexts while meeting the deadlines of all the jobs. For a hard real time systems, the problem can be stated as an optimization problem as follows:

*Schedule the workload such that all jobs meet their deadlines and **cost(H)** is minimized where the cost function $cost(t)$ at time t is defined as follows:*

$$cost(t) = \begin{cases} 0 & \text{if } t = 0 \\ cost(t-1) & \text{if } \Omega(t) = \Omega(t-1) \\ cost(t-1) + 1 & \text{if } \Omega(t) \neq \Omega(t-1) \end{cases}$$

Since the optimization metric is context-dependent, standard optimization techniques cannot be used to solve the problem. Making all the jobs non-preemptive does not solve the problem either. If all the jobs are non-preemptive, there could still be $J - 1$ switches in the worst case where J denotes the number of jobs of Γ in H. One can imagine a preemptive schedule having a lot of preemption between jobs

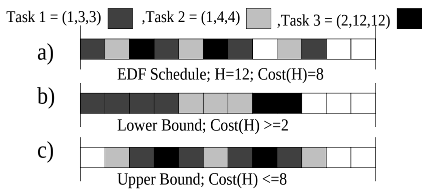

Figure 1. A set of 3 periodic tasks

of the same context but very few *context switches*. A non-preemptive schedule minimizes the number of preemption in a schedule while a context-aware schedule seeks to minimize the number of times the scheduler changes *context*.

2. Performance Bounds

In this section, we present bounds on the maximum and minimum number of switches produced by any feasible scheduling algorithm.

Lemma 1 *There will be at least* $n - 1$ *switches every* $\max_{1 \le j \le n} T^j_{min}$ *units of time where* $T^j_{min} = min\{T_i | \ \tau_i \in \Gamma_j\}$

Proof Sketch: Each context j has to execute *at least once* every non-overlapping window of time of size T^j_{min}. Otherwise the minimum-period task of Γ_j will miss its deadline. Therefore, all n contexts will execute at least once every non-overlapping window of size $\max_{1 \le j \le n}(T^j_{min})$. Given this fact, there has to be at least $n - 1$ switches in each of these windows.

Lemma 2 *The maximum number of switches* S^n_{max} *that can occur in the hyper-period of a schedule is given as follows:*

$$S^n_{max} = \begin{cases} \sum_{j \le n} x_j - 1 & if\, x_n \le \sum_{j<n} x_j \\ 2\sum_{j<n} x_j & if\, x_n > \sum_{j<n} x_j \\ 0 & if\, n = 1 \end{cases}$$

where x_j *denotes the time jobs of* Γ_j *execute in* H *and the contexts are ordered such that* $x_1 < x_2 < ..... < x_n$

Proof Sketch: The above equation corresponds to the pathological case when a different context starts executing at the beginning of each time unit as long as some different context job is available to preempt It can be proved inductively. The base case is obvious. There will be no switches if there is only one context. Assume S^{i-1}_{max} is correctly calculated by the formula. Since context i executes for x_i time in the hyper-period, it can be thought of as having x_i blocks of the same color where one time unit is a block. What we have to basically do is to place the x_i blocks among the existing $\sum_{j<i} x_j$ blocks in such a way that the switchings are maximized. The proof is omitted here due to space constraints.

Figure 1(b) shows the performance bound of the task set. Since there are three contexts and each context has only one job, $max(T^j_{min}) = max(T^j) = 12$. This means there will definitely be at least 2 context switches every 12 units of time. For the upper bound, we have 4 blocks of context 1, 3 blocks of context 2 and 2 blocks of context 3. The maximum number of switches for such a configuration is 8 as shown in the figure. Once can see from the example is that EDF scheduler is not very context-friendly.

The lower bound of Lemma 1 does not consider the fact that jobs cannot execute at any point in time but only during the interval between their release time and deadline. Incorporating this constraint would make the bound even tighter. This is part of on-going research.

3. Offline Solution

Cyclic Schedule: Constructing a feasible off-line schedule for a task set is a well-studied problem [4]. A cyclic structure is used and scheduling decisions are made periodically. The scheduling decision times partition the time line into intervals called frames. Every frame has length f; the frame size. F denotes the number of frames in the hyper-period.

Given the task set and the frame size, the cyclic schedule (if one exists) can be constructed using the well-known network formulation of the preemptive scheduling problem [3]. All the (J) jobs and (F) frames within the hyper-period of the schedule are represented as nodes of a graph, along with two special nodes called source and sink. Edges exist between the source and all the job nodes and between all the frame nodes and the sink. The capacity of edges joining the source node to the Job nodes is equal to the execution time of the job whereas the capacity of all the other edges is equal to the frame size. The edges from the source to the job nodes can be thought of as the demand (computation time) of the system while the edges from the frames can be thought of as the supply (time). The constraint is expressed as follows. An edge exists between a Job node and a Frame node if and only if the interval of the frame is fully contained within the feasible interval of the job.

Many algorithms exist for finding maximum flow of a network-flow graph [1]. The maximum flow of the network-flow graph defined above can be at most equal to the sum of the execution times of all the J jobs. If the maximum flow is indeed equal to the sum of execution times, then the flow of the graph tells one how to decompose the tasks into subtasks and which subtask to schedule in which frame. If the maximum flow of a graph is less, then no feasible cyclic schedule exists with the frame size used to generate the graph. In this case, a smaller frame size is chosen and the network flow algorithm repeated till a feasible schedule is found. In the worst case, the frame size could become 1 in which case this would reduce to brute force.

Some schedulability is lost as the jobs are not eligible for execution for their entire feasible interval but only in the set of frames which are fully contained within the job. If the task set is such that the release time and deadlines of all jobs coincide with the frame boundaries, then no schedulability is lost. For instance, this would be the case when the frame size divides all the periods and the release time and

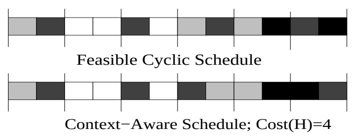

Figure 2. Fiddling with the intra-frame jobs.

deadlines of all the jobs are at the beginning and end of the period respectively. For a general frame size however, some schedulability might be lost. However, for task sets with reasonable amount of slack (which is the focus of this paper), this approach would work reasonably well.

Context-Aware Schedule: The key point of the network-flow algorithm which makes it so appealing from a context-aware scheduling point of view is that the schedule generated consists of jobs whose deadlines are always *after the end of the frame within which they are being scheduled*. Therefore, the order of execution of jobs within a frame can be freely changed to produce a resource-friendly schedule, without the fear of any job missing its deadline. Figure 2 shows a cyclic schedule for the same example produced by the network-flow algorithm. After fiddling with the order, there are only 4 switches in the hyper-period.

Clearly, a bigger frame size allows for more possible clustering opportunities and is thus better in terms of performance(cost(H)). Unfortunately, as discussed before, the bigger is the frame size, the more are the chances of the task set not being schedulable. The set of constraints governing the frame size have been well studied in [4]. To ensue that there exists at least one eligible frame for each job, we need

$$2f - gcd(T_i, f) \leq D_i \;\; \forall i \tag{1}$$

In the special case when release time of all the jobs coincide with the frame boundaries, (for instance, this would be the case when the minimum period divides all the other periods) it suffices to choose a frame size that is equal to or smaller than D_i for all i. Furthermore, if the minimum deadline D_{min} divides all the larger deadlines, there would be no schedulability loss.

To keep the length of the cyclic schedule as short as possible, the frame size f should be chosen so that it divides H. This condition is met when

$$\lfloor T_i/f \rfloor - T_i/f = 0 \tag{2}$$

holds for at least one i i.e when the frame size divides the period of at least one task.

Performance: From the constraints described above, one can see that the maximum possible frame size for any task set is at most the minimum deadline D_{min}. Using a similar logic that we applied for the window of time in Lemma 1, once can see that there can be at most $n-1$ switches inside each frame. Which means that for task sets which are feasible with the maximum frame size, there will be *at most* $n-1$ *switches every* D_{min} units of time. For task sets in which all the jobs have deadlines at the end of the period ($D_i = T_i$), there will be a a maximum of $n-1$ switches every T_{min} units of time The actual number of context switches could be less, depending on the how many jobs of each context are present in each frame. Lemma 1 however, tells us that there could never be less than $n-1$ switches every $\max\limits_{1\leq j\leq n} T^j_{min}$ units of time.

4. Online Solution

On-line solution is difficult: The model described in section 1 is restrictive in the sense that precise knowledge of all the task parameters is generally not known beforehand. The number of tasks is also assumed to be fixed. A more realistic scenario would be for the scheduler to have bounds of the minimum inter-arrival time with new tasks entering and existing tasks leaving the system at arbitrary time instants. An on-line scheduler, therefore, typically makes scheduling decisions based on the current state of the system and does not on the possible release times of future jobs. This makes on-line context aware scheduling quite difficult.

Lemma 3 *It is impossible for a scheduler to produce an optimal schedule without knowing release times of future jobs.*

Proof Sketch: This can be seen by an adversarial argument as follows: Consider 2 contexts for simplicity. At any time t, suppose that a color 0 job (by a color i job, we mean a job belonging to context i) has finished but there are color 1 jobs with considerable slack in the ready queue. If the processor switches to a color 1 job at time t, at time $t+\delta$, a color 0 job with no slack could arrive making the processor switch back and thus making the switch at time t useless. On the other hand, if the processor does not switch at time t and stays idle, there could be a case when a huge number of jobs arrive at time $t+\delta$ making the old color 1 jobs at time t miss their deadline.

Heuristic: Let us consider only two contexts for simplicity. Since the deadlines of all the jobs have to be met anyway, a reasonable approach could be to build the context-aware scheduler over existing optimal (in terms of deadlines) algorithms like EDF [2]. The EDF (or some other optimal algorithm) scheduler could be run as usual as long as it schedules jobs of the same context. The time instant when EDF switches contexts is the time when a decision needs to be made i.e should the context-aware scheduler switch as well or should it continue with jobs of the currently executing context. Consider the following qualitative argument (translating the qualitative argument into equations is part of on-going work). Here X denotes the context of the currently executing job belongs to and Y is the other context.

If the existing context X has a lot of slack and the Y job has very little slack, clearly, it makes sense to switch immediately to the Y job (The X jobs can wait while Y jobs cannot. Moreover, the Y jobs can run for a while without being bothered by the X jobs). By a similar argument, if the existing context X jobs have very little slack and the Y job has a lot of slack, then it makes sense to block the Y job. them for a while and continue running the X context jobs.

If both context X and Y jobs have a low slack, then the slack information is not enough to make a decision. It might make more sense, however, to chose a context whose sum of remaining execution times of the current jobs is lower. This is based on the reasoning that since one cannot ignore low-slack jobs for a long time anyway, it might be best to finish them off as quickly as possible.

If both context X and Y jobs have a lot of slack, then choosing the context with the longer remaining execution time might be a better option. The reasoning behind this is that since both applications have a lot of slack, deadlines are not an issue (in the near future). Thus choosing the context with the longer remaining execution time would make the processor run on one color for a longer time.

There are a lot of technical details to be filled in the above qualitative argument. This is part of on-going work.

5. On-Going Research

In this section, we outline ongoing work for solving the online approach. There are many questions that needs to be answered for example what constitutes a *high* or a *low* slack? What is a safe time to admit a new task? If the context-aware scheduler makes a scheduling decision that is different from EDF, how long can it block the Y context job without compromising any deadline constraints? Since the scheduler makes a different decision, there is no longer any guarantee that all future deadlines will surely be met.

For the last question, there has been a lot of work in the real-time community in solving similar problems. We can adapt them to solve our problem as follows.

Critical sections: Effectively, what we are doing is delaying the execution of higher priority jobs with earlier deadlines in favor of jobs of the same context. This situation is analogous to a higher priority job getting blocked by a lower priority job which is executing in its non-preemptive critical section. In this case, the higher priority job is never allowed to preempt the lower priority job while in our case, the higher priority job can preempt but at a cost which we would like to minimize. A lot of work has been done in analyzing the schedulability of a task set with critical sections. We can use this analysis in the reverse direction, i.e. given the task set, compute the longest possible critical sections for each task which still ensures schedulability of the system. When the context-aware scheduler decides to block a job, we could then *pretend that the currently executing job has just entered its critical section* and thus block the Y context job for the length of its critical section.

Aperiodic jobs: Another way to solve this problem could be to use the analysis of aperiodic jobs. Normally, aperiodic jobs are executed ahead of periodic jobs in the system as long as they don't cause any real time task to miss its deadline. This is similar in spirit to what we want to do i.e *schedule jobs of the same context in clusters as long as they don't cause the other jobs to miss their deadline*. Therefore, one way to solve our problem would be the following: When EDF switches context, pretend an aperiodic job has entered the system at that time. *Block the higher priority job for as long as the aperiodic job would have been executed by the scheduler.* The higher priority job would not miss its deadline since the aperiodic scheduler never causes any job to miss its deadline. Many such aperiodic schedulers exist with varying performance and complexity. *Slack-Stealing algorithms* [9] can tell you exactly how long one can block a higher-priority job without missing its deadline but has a very high overhead. *Period Servers* [5] [8] offer simplicity but their performance is not as good while *dual priority* [6] lies somewhere in the middle.

6. Related Work

To the best of our knowledge, no work has yet been done on context-aware real-time scheduling, as we define it in this paper. On a more basic level, however, our problem amounts to exploiting the slack of the system so as to achieve ancillary goals (in addition to meeting deadlines). There has been much work along these lines; space constraints prevent us from adequately portraying this body of work. Representative examples include the use of slack-stealing for improving precision [10], and more recently for minimizing power consumption (by trading off slack for processor speed).

Perhaps the closest piece of work to ours is the concept of preemption threshold proposed by Saksena and Wang [7] to minimize the number of thread preemption and CPU context switches. The problem we consider in this paper differs in the sense that what constitutes a change in context is *explicitly* spelled out as opposed to being implicitly assumed for *every* job preemption.

References

[1] A.V.Goldberg. Processor efficient implementation of a maximum flow problem. 38:179–185, May 1991.

[2] C.L.Liu and Layland. Scheduling algorithms for multiprogramming in a hard real-time enviroment. 20:46–61, 1973.

[3] J.Blaswicz. Selected topics in scheduling theory. 31, 1987.

[4] J.W.S.Liu. Real Time Systems. 2000.

[5] M.Spuri and G. Buttazo. Scheduling aperiodic jobs in dynamic priority systems. 10:179–210, 1996.

[6] R.Davis and A.Wellings. Dual priority scheduling. December 1995.

[7] M. Saksena and Y.Wang. Scalable Real-Time System Design using Preemption Thresholds . 2000.

[8] T.M.Ghazalie and T.P.Baker. Aperiodic servers in deadline scheduling enviroments. 9:31–68, 1995.

[9] T.S.Tia. Utilizing slack time for periodic and sporadic requests in real time systems. (UIUCDCS-R-95), April 1995.

[10] W.K.Shih and J.W.S.Liu. Minimization of the maximum error of imprecise computations. March 1995.