SNS: A Simple Model for Understanding Optimal Hard Real-Time Multiprocessor Scheduling

Greg Levin, Caitlin Sadowski, Ian Pye, Scott Brandt
Computer Science Department
University of California, Santa Cruz
{glevin, supertri, ipye, sbrandt}@soe.ucsc.edu

Abstract

We consider the problem of optimal real-time scheduling of periodic tasks for multiprocessors. A number of recent papers have used the notion of fluid scheduling to guarantee optimality and improve performance. In this paper, we examine exactly how fluid scheduling techniques overcome inherent problems faced by greedy scheduling algorithms such as EDF. Based on these foundations, we describe a simple and clear optimal scheduling algorithm which serves as a least common ancestor to other recent algorithms. We provide a survey of the various fluid scheduling algorithms in this novel context.

1. Introduction

Multiprocessors are becoming commonplace as more and more computers, even desktops, have multiple cores. Many of the implications of a multiprocessor system are still not well understood, and multiprocessor scheduling is one of them. Multiprocessor scheduling is particularly difficult in the presence of real-time constraints. Most previous real-time multiprocessor scheduling algorithms (e.g., [5, 14]) use partitioned scheduling: tasks are statically allocated to processors, then local schedulers on each processor are responsible for ordering the tasks assigned to that processor. Partitioned approaches are easy to implement, as they reduce multiprocessor scheduling to uniprocessor scheduling. However, they are not optimal, in the sense that they can fail to schedule theoretically feasible task sets[1].

In 1996, Baruah [4] introduced the pfair algorithm, the first optimal multiprocessor scheduler for periodic tasks. By migrating tasks between processors, pfair can successfully schedule any task set whose utilization does not exceed processor capacity. More recently, a number of papers have exploited the notion of *fluid scheduling* to achieve optimality while greatly reducing the number of required context switches and process migrations [3] [6]. Subsequent papers have expanded on these basic models. All these algorithms, while superficially different, have achieved optimality by expanding on a single core idea. A better understanding of this common framework and why it succeeds will aid in the ability to understand, compare, and contrast these algorithms; to consolidate their insights; and point towards new avenues of study.

The contributions of this papers are as follows:

- We examine the difficulties of optimal hard real-time multiprocessor scheduling, and discuss the feasibility of a truly optimal scheduler
- We give a simple set of sufficient conditions for optimal scheduling and show how the implied scheduling model is actually the foundation of recent scheduling algorithms.
- We provide "Stack-and-Slice" (SNS), the simplest optimal scheduler to date[2], with reasonably good migration bounds and almost no computational overhead.
- We survey and explain recent scheduling algorithms relative to our new framework.

In Section 2 we formalize the problem under consideration. In Section 3, we will consider greedy scheduling algorithms, introduce the notion of *dual schedules*, and examine why greedy algorithms tend to fail in a multiprocessor environment. In Section 4, we will see how approximating fluid schedules can overcome these difficulties, and prove simple sufficient conditions for optimality. We use these conditions to introduce our SNS algorithm. In Section 5, we examine recent algorithms in more detail, and compare their strengths and weaknesses. In Section 6, we suggest several avenues of future study.

[1] In fact, examples may be constructed where partitioned schedulers fail to successfully schedule tasks sets that only require $(50 + \epsilon)\%$ of processor capacity [3].

[2] SNS is similar to EKG [3]

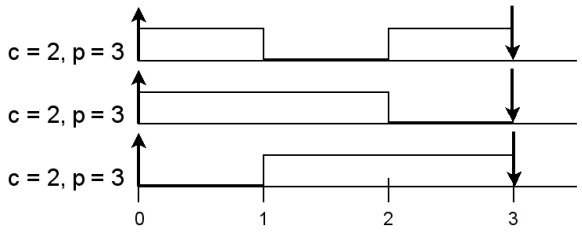

Figure 1. Three tasks, each with a rate of $2/3$, can run successfully on two processors with migration.

2. Background

We assume a multiprocessor system with M identical CPUs. Given a collection of periodic tasks (processes), the basic scheduling problem is to determine which task to run on each processor at any given instant, with the restriction that no task can run on multiple CPUs at the same time. Each task needs a certain amount of CPU time to complete its workload within each period. A *feasible* schedule is one where all such time constraints are met.

Not all feasible schedules are equally good; scheduling algorithms must also try to minimize the number of context switches (preemptions) and process migrations. Both these operations incur overheads. Although highly system-dependent, task migrations generally take longer than context switches, sometimes prohibitively longer, so that partitioned schemes are preferred in practice. Ironically, all algorithms we consider make the simplifying assumption that both context switches and migrations are "free," even though they strive to minimize these operations. To better explore the same theoretical framework as these algorithms, we will adopt the same convention, although results may not be achievable on a real system.

Given this assumption of no overhead, were feasibility our only goal, it would not matter *which* processor was hosting a given task, only which tasks were running at a given time. This assumption can lead to clearer scheduling descriptions (e.g., Figures 1, 3, 4, 5 in this paper). In fact, some recent algorithms give no explicit prescription for how to assign tasks to processors [6], [11].

We will assume a fixed workload of N periodic tasks, denoted $T_1, \ldots, T_N$. Task T_i has period p_i, and workload (or required "on" time) c_i within each period. We refer to a task's single period workload as a *job*. A task's release time (the time when it becomes schedulable) coincides with the deadline of the previous period. A task's *rate*, or *utilization*, is its workload divided by its period: $u_i = c_i/p_i$; that is, what fraction of its life must be spent running. We wish to schedule our tasks so that each task completes its required workload during each of its periods. The *total rate* of a set of tasks is the sum of the individual rates for each task. Tasks may be represented succinctly as the pair (c_i, p_i).

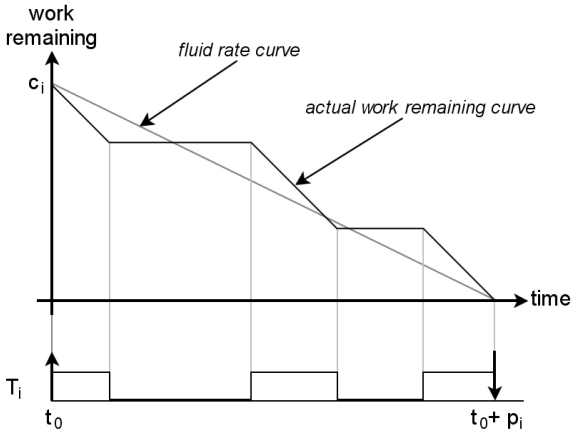

Figure 2. Fluid versus Practical Schedules

For now we will assume no scheduling overhead, so that we should have enough CPU time to complete all jobs (i.e., a feasible schedule exists) provided:

(i) Total task workload doesn't exceed total CPU capacity ($\sum_i u_i \leq M$).

(ii) No task's workload exceeds its period ($u_i \leq 1 \ \ \forall i$).

(iii) Process migration is allowed.

We say that a set of tasks is *feasible* if these conditions are met, and a scheduling algorithm is *optimal* if it can successfully schedule any feasible task set. A simple example depicted in Figure 1 demonstrates a set of 3 tasks that can be successfully scheduled on two processors only when one of them divides its time between both CPUs. We will consider the case where processes are free to migrate between CPUs whenever necessary.[3]

Given unlimited context switching and migration, it is not hard to see that the first two constraints are sufficient in our theoretical model. In fact, this is just an extension of the uniprocessor case presented by Liu and Layland [15]. Imagine that we can reschedule our jobs after each length ϵ unit of time. As $\epsilon \to 0$, we can turn each task T_i on or off sufficiently often so that it appears as if it is constantly running on only a fraction u_i of a processor. In the limit, each job executes at exactly its necessary rate and, when all rates sum to no more than M, all jobs will finish on time. Cho et. al. [6] refer to this as a *fluid* scheduling model. Figure 2 shows the difference between a fluid and an actual scheduling of a task. We can easily determine whether a set of periodic tasks may be feasibly scheduled; much more challenging is actually *finding* a feasible schedule that minimizes context switches and migrations. This has been the goal of recent papers on this subject and is our primary interest.

[3] Without migration, optimal scheduling reduces to the NP-Hard bin packing problem [9].

3. What's Wrong with Greedy Schedulers?

3.1. Greedy Scheduling Algorithms

An attractive (and common [5]) first approach to scheduling is to try to find a simple greedy solution. Greedy algorithms are straightforward to explain, prove and implement. They often attempt to encapsulate the criticality (likeliness of a missed deadline) of a job into a single number and then greedily schedule those jobs based on that.

Several successful greedy algorithms have been found for uniprocessor scheduling. Two of the earliest are from a seminal paper by Liu et. al. [15]. The first is Rate Monotonic (RM) scheduling, which statically sorts jobs by their rates, and guarantees feasible schedules for up to $\sim 70\%$ processor utilization. 100% utilization in guaranteed by their second algorithm, Earliest-Deadline-First (EDF). Here, whenever jobs are completed or introduced, the job with the earliest deadline is selected to run. While EDF is optimal on a uniprocessor, it is suboptimal in a multiprocessor environment [8]. A task set where EDF fails to find a feasible schedule is depicted in Figure 1.

Another well studied scheduling algorithm which is optimal on one processor is preemptive Least-Laxity-First (LLF), initially introduced as the least slack algorithm [16]. LLF always runs the job with least *laxity*: time remaining until the next deadline minus time required for the remaining workload (i.e., allowable idle time). Since LLF requires laxity to be recomputed at every clock tick, it has a large overhead, and causes numerous context switches when two jobs have the same laxity. MLLF [17] reduces scheduling events and context switches, but is still more complicated than EDF and provides no obvious advantage. Although neither LLF nor MLLF are optimal in a multiprocessor setting ([14] and Figure 3), MLLF can find a feasible schedule in some cases where EDF cannot (again, see Figure 1).

The LLF scheduler is based partially on the observation that a schedule has become infeasible if any job ever has negative laxity in its current period (a job has more work than time remaining). Clearly, any job whose laxity has reached zero must be immediately activated and run continuously until its deadline in order to complete its work on time. This leads us to our second consideration when designing a greedy algorithm. The "greedy" part tells us *which* tasks to schedule, but we must also specify *when* we will do the scheduling. That is, at what times should we re-sort and apply our greedy preference? We have a list of three *standard scheduling events*:

A. RELEASE**:** A task has reached the end/beginning of its period

B. WORK COMPLETE**:** A task has finished its work for its current period, and must be turned off

C. ZERO LAXITY**:** A task has no remaining laxity for its current period, and must be turned on

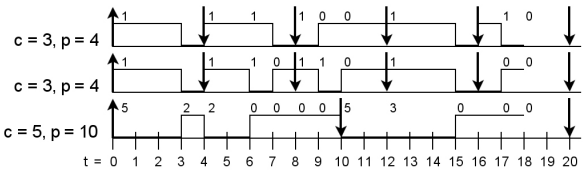

Figure 3. LLF Fails for Multiprocessors
Two processors, and three jobs that require 100% utilization. Numbers above each schedule indicate a task's laxity at a scheduling event, with numbers absent from completed jobs. An idle processor at time 3 eventually leads to failure.

Any simple greedy algorithm will specify a greedy sort key and which scheduling events it will observe. For example, when ZERO LAXITY events are added to EDF, we get a hybrid scheduler known as EDZL [7]. While this provides an improvement over standard EDF for multiprocessors, it is still not optimal.

3.2. Schedule Duality

To improve upon greedy schedulers, it is necessary to understand why they fail. To aid with this, we introduce the notion of a "dual schedule." Consider the feasible task set

$$\{\ T_1 = (3,4),\ T_2 = (3,4),\ T_3 = (5,10)\ \}$$

shown in Figure 3; LLF cannot successfully schedule this job set on two processors. T_1 and T_2 each have laxity of 1, while T_3 has laxity of 5, so LLF would initially schedule jobs T_1 and T_2. If these are run to completion, the laxity of T_3 will have only gone down to 2 (> 1), so it would never be chosen over T_1 or T_2 before they are finished, regardless of any preemptive scheduling events. When T_1 and T_2 finish at $t = 3$, T_3 is the only unfinished task remaining. One processor stays idle until T_1 and T_2 are re-released at $t = 4$. This idle time ensures the scheduling's eventual failure[4], even though the failure is not evident until all three tasks reach zero laxity at time 18. The following theorem generalizes this observation.

Theorem 1 *When the total rate of a task set is equal to the number of processors and all tasks have the same initial release time of $t = 0$, then no feasible schedule can allow any processor to remain idle for any length of time.*

Proof. Given tasks $T_1, \ldots, T_N$ on M processors where the rates sum to M. In a feasible schedule, task T_i, at the end of k periods, must have done work equal to $kc_i = k(p_i u_i) = (kp_i)u_i = t_k u_i$, where t_k is the ending time of the k^{th} period. Let t_a be the first positive time at which all tasks reach a deadline simultaneously (i.e., the least common multiple

[4]Between $t = 4$ and $t = 20$, $(2 \times 4 \times 3) + (4 + 5) = 33$ units of work must be done, though the two processors can only accomplish 32.

of their periods). Then the total work done by all tasks at time t_a must be $\sum_i t_a u_i = t_a \sum_i u_i = t_a M$. This much work can be accomplished by time t_a only if all processors are running continuously until this time. Any idle time implies less than $t_a M$ total work is done, and the scheduling fails. □

We can now see where LLF failed on our previous example. While T_1 and T_2 were being prioritized due to their high required workloads, their required idle time (1 unit idle time out of the 4 unit period) was not being considered. Consequently, when they finished, they both had idle time to use up before the next RELEASE. However, an optimal scheduling could only consume idle time from one task at a time, since two tasks need to be on at all times to satisfy Theorem 1. By distributing their idle times so as not to coincide (Figure 4), this task set can be easily scheduled. This observation may be formalized by the notion of a *dual schedule.* Given the problem of scheduling tasks

$$\{\ T_1 = (c_1, p_1),\ \ldots,\ T_N = (c_N, p_N)\ \}$$

on M processors, the *dual scheduling problem* consists of

$$\{\ T'_1 = (p_1 - c_1, p_1),\ \ldots,\ T'_N = (p_N - c_N, p_N)\ \}$$

on $N - M$ processors. Given any scheduling (feasible or otherwise) of $T_1, \ldots, T_N$, the *dual scheduling* of $T'_1, \ldots, T'_N$ is the schedule that has T'_i turned on precisely when T_i is turned off (and vice versa).

Any job has a certain amount of work that it must complete by some deadline. Reciprocally, it must also achieve a certain amount of idle time by that deadline. When the total utilization of all jobs is M, scheduling the idle time is just as important as (in fact, is equivalent to) scheduling the work time. The dual problem makes this explicit: laxity in the primal (original) problem is work remaining in the dual, and vice versa. A WORK COMPLETE event in the primal is a ZERO LAXITY event in the dual. Since we must have M processors constantly running in the primal, there must always be $N - M$ idle tasks, and so our dual problem has $N - M$ processors. The task set

$$\{\ T_1 = (3, 4),\ T_2 = (3, 4),\ T_3 = (5, 10)\ \}$$

on two processors, and its dual task set

$$\{\ T_1 = (1, 4),\ T_2 = (1, 4),\ T_3 = (5, 10)\ \}$$

on one processor, are shown in Figure 4. Note that the dual of a dual is just the primal (as is the case with any good notion of "duality"). Also note that the feasible scheduling shown is achieved by applying LLF to the dual problem.

An interesting special case of duality yields:

Corollary 2 *Any scheduling problem with M processors and $M+1$ tasks (where the total rate of the tasks is M) may be scheduled by applying EDF to the uniprocessor dual.*

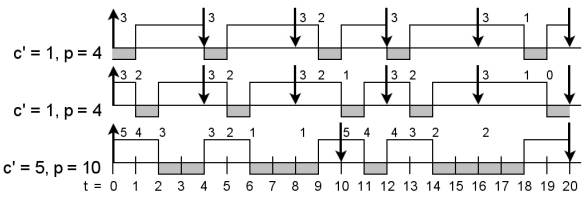

Figure 4. A Schedule and Its Dual
This shows the previous task set, scheduled using LLF on the dual. The dual (idle) task running is shown in gray below each time axis, with dual laxity (i.e., work remaining) at each scheduling event shown above.

Proof. This follows directly from the above discussion and the fact that EDF is optimal for uniprocessor scheduling. □

From the primal's view, the above translates into the following scheduling policy:

1. Schedule all tasks *except* the one with the earliest deadline
2. The only preemptive scheduling events needed are ZERO LAXITY and RELEASE times

It seems strange from the primal's point of view, but we will never need to explicitly schedule WORK COMPLETE events, any more than we need to schedule ZERO LAXITY events for EDF on a uniprocessor; if the task set is feasibly schedulable, these will attend to themselves.

One course of inquiry suggested by duality is a search for *dual-symmetric* schedulers: scheduling algorithms that produce equivalent results when applied to either the primal or dual of a scheduling problem. EDF and LLF are *not* dual-symmetric. As shown in the preceding discussion, EDF turns *on* the tasks with the earliest deadlines, whereas EDF applied to the dual would turn *off* these earliest deadline tasks. Similarly, LLF turns on tasks with the least laxity, while LLF on the dual turns off tasks with the least work remaining; while these can lead to the same schedules in some cases, they are fundamentally different orderings of tasks.

An example of a dual-symmetric scheduler would examine the *difference* between remaining work and laxity, and apply a greedy scheduling by sorting according to this difference (either in absolute terms or as a ratio). Tasks with larger work would be preferentially turned on, and tasks with higher laxity would be preferentially turned *off*, until no more "on" or "off" slots were available. Note that this algorithm treats work remaining and laxity in a symmetric fashion, and so is dual-symmetric, as is SNS, developed in the next section.

While such dual-symmetric schedulers address one specific scheduling problem, they fail to address another:

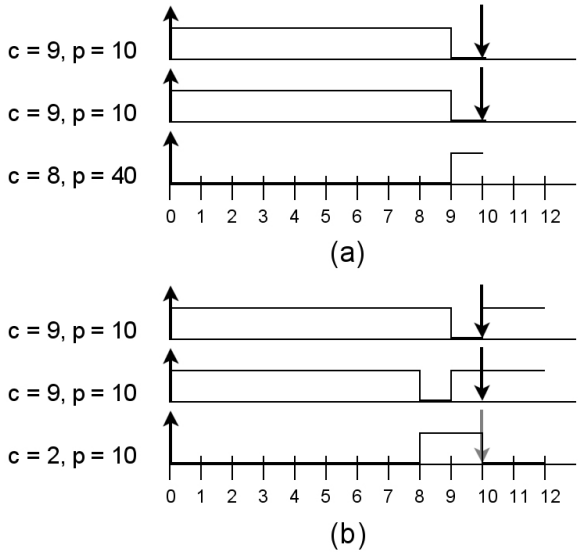

Figure 5. Greedy Counter-example
Task set which confounds most greedy schedulers using common events. (a) shows an incorrect greedy scheduling, while (b) shows a feasible proportional scheduling.

global knowledge. Consider the following task set with two processors:

$$\{ T_1 = (9, 10),\ T_2 = (9, 10),\ T_3 = (8, 40) \}$$

It seems inevitable that any greedy scheduling policy would choose to schedule tasks T_1 and T_2; which have earlier deadlines, smaller laxities, and high work/laxity ratios, over task T_3; which has huge laxity, low rate, and a long deadline. Of the three types of scheduling events we've considered so far– RELEASE, WORK COMPLETE, and ZERO LAXITY– if we start by turning T_1 and T_2 on, the first event is the WORK COMPLETE event at time 9. At this point, we know our schedule is infeasible because we are left with an idle processor for one time unit (see Figure 5a).

Our conclusion is that the {A, B, C} set of scheduling events is inadequate. A critical juncture occurs at time 8 when either T_1 or T_2 *must* be turned off. After that, the *collective* idle time required by T_1 and T_2 prior to their deadline exceeds the idle capacity of the system. This can be seen only by considering T_1 and T_2 *as a set*; no criterion applied to individual tasks can reveal this fact. Must all possible subsets be examined for the scheduler to "see" this event, leading to an NP-Hard search space? Or can it be revealed by some simpler global property on all tasks? While finding a minimum necessary and sufficient set of scheduling events remains an open problem, recent advances have revealed a sufficient superset of events (Section 4).

4. Optimality Via Proportional Fairness

To date, the only known solutions to the "global knowledge" problem are variations on the notion of *proportional fairness*. By overconstraining the problem, proportional fairness forces tasks to march in step with their fluid rate curves more precisely than is theoretically necessary. Suppose we modify our previous example to

$$\{ T_1 = (9, 10),\ T_2 = (9, 10),\ T_3 = (2, 10) \}\ .$$

All we have done is to impose the additional requirement that task T_3 complete a proportional share of its work every time the other tasks hit their deadlines. Suddenly we have a ZERO LAXITY event at time 8. T_3 will then be switched on; T_1 and T_2 will each run for one of the remaining two time units on the other processor, and a feasible schedule results (see Figure 5b). In this example, where the third period was a multiple of the first two, it is easy to reformulate the problem in this way. When we have numerous jobs with periods which could be relatively prime, the question of when to force jobs to hit their proportional rate quotas becomes more complicated.

The first solution to this problem was the pfair scheduling scheme [4]. pfair creates a scheduling event and recomputes the set of running tasks at every multiple of a discrete time quantum. The notion of proportional fairness used is very strict, requiring the actual work completed by a task to be within 1 unit of its fluid rate curve at each time quantum. The result of this policy is a large number of scheduling calculations and context switches, with correspondingly high overhead. Intuitively, it seems unnecessary to adhere so closely to the fluid schedule: performance could be improved by a more judicious choice of scheduling events.

4.1. Simple Conditions for Sufficiently Fluid Schedules

Several other papers [3] [6] have observed that fluid schedules need not be followed so rigorously. Any given job really only needs to match its fluid rate curve at its own deadline. However, it is difficult to plan for the added complexity imposed by the presence of multiple jobs simultaneously running on different processors. A sufficient (although not necessary) compromise is to require *all* tasks to lie on their fluid rate curves at the deadline of *each* task. In this model, time is partitioned into *windows*, demarcated by all the deadlines of all tasks in the system. Within each window, all jobs still have their usual rate, but now all jobs share the same deadlines. While this new requirement overconstrains the system, it also greatly simplifies the scheduling process.

Suppose that the length of the time window between some two deadlines is t_f. In this restricted view, we will consider time to start at $t = 0$ and end at $t = t_f$. Task T_i

must do work equal to $u_i \times t_f$ during this time window; in the view of Figure 2, its work remaining curve must go from $u_i \times t_f$ to 0. As all rates sum to no more than M, we have no more than $M \times t_f$ total work to do during this interval. We simply have to decide when to turn jobs on and off so that their work remaining curves meet their goals.

The shortening of local deadlines will create new scheduling events (e.g., the new ZERO LAXITY event at $t = 8$ in Figure 5b). The WORK COMPLETE and ZERO LAXITY events that occur based on the local deadline at the end of a time window will be referred to as *secondary scheduling events*. These events have the usual scheduling implications: zero laxity jobs must be turned on and completed jobs must be turned off. As we shall see, this new set of scheduling events is sufficient for optimal scheduling (i.e., is a superset of the minimal necessary and sufficient set of events previously mentioned).

For the remainder of the discussion of our model, we will make the simplifying assumption of full utilization. Since our primary focus is optimal scheduling algorithms, we will assume that the sum of the utilizations of all tasks is M. If this were not the case, then one or more dummy (idle) tasks with an arbitrary period could always be added to make up the difference. These would represent idle processor time. Assuming full utilization simplifies the following discussions and loses no generality.[5]

By partitioning time into windows so that all tasks have the same (local) deadline, we have created very simple necessary and sufficient conditions for ensuring a feasible scheduling.

Theorem 3 *If a set of jobs (with total utilization M) can be feasibly scheduled within a time window, then any scheduling policy will find such a feasible schedule if and only if it observes the following 3 rules for local workloads:*

1. *Always run any job with zero laxity*
2. *Never run any job which is completed*
3. *At any moment, M distinct jobs must be running* □

That these conditions are necessary is clear: if laxity for a job becomes negative, it cannot be completed on time; running a completed job is locally equivalent to an idle processor, and an idle processor means that less than a total of $M \times t_f$ work will be done in the window, implying failure. We now show that these conditions are also sufficient.

Lemma 4 *A scheduling of jobs which follows Rules (1)-(3) will fail to complete all tasks on time if and only if, at some time before t_f, there are (at least) $M + 1$ jobs with zero laxity.*

[5]In practice, there are many productive ways to take advantage of the free processor time resulting from sub-100% utilization (Section 5).

Proof. $M+1$ jobs with zero laxity cannot all be finished on time on M processors. On the other hand, the set of zero laxity jobs can only grow as time increases, and any job which has not hit zero laxity by time t_f will be completed before then. If there are at most M jobs with zero laxity as time reaches t_f, the other $N - M$ jobs will be completed. Rule (1) ensures that up to M zero laxity jobs may be run simultaneously and to completion. Therefore, an $(M+1)^{st}$ zero laxity job is necessary for failure. □

The point at which that $(M+1)^{st}$ job reaches zero laxity is referred to as a *critical moment*.[6] In the absence of critical moments, the M processors do the required amount of total work, and all tasks will be completed on time. We now introduce some convenient notation, deviating a bit from [6]. Anything subscripted with i, j will indicate job i at the j^{th} secondary event (which we'll assign time t_j). $c_{i,j}$ is the *local work remaining* to job i at time t_j. $u_{i,j} = \frac{c_{i,j}}{t_f - t_j}$ is the *local remaining utilization* of job i, namely, the average rate at which it must consumed to successfully complete. Finally, we define $S_j = \sum_{i=1}^{N} u_{i,j}$ to be the *total local utilization* at time j, (the rate at which the whole system must run).

Lemma 5 (Cho et. al.) *If $S_j \leq M$ at time t_j, then event j is not a critical moment.*

Proof. If event j is a critical moment, then each of the $M + 1$ zero laxity jobs (for simplicity, let's call them $T_1, \ldots, T_{M+1}$) has $u_{i,j} = 1$. Then $S_j = \sum_{i=1}^{N} u_{i,j} \geq \sum_{i=1}^{M+1} u_{i,j} = M + 1 > M$. □

In order to show that a scheduling following Rules (1)-(3) is feasible, it suffices to show that $S_j \leq M$ at each secondary event j.

Proof of Theorem 3. If we let W_j be the *total work remaining* at time t_j, then

$$W_j = \sum_{i=1}^{N} c_{i,j} = \sum_{i=1}^{N} u_{i,j}(t_f - t_j) = (t_f - t_j)S_t \ .$$

According to Lemma 5, we can only run into trouble if S_j exceeds M at some event. However, at any time t_j within the scheduling window, if all M processors have been fully utilized between times 0 and t_j, then W_j has been getting reduced at a constant rate of M-per-time unit. That is, $W_j = W_0 - t_j \cdot M$. Since $W_0 \leq t_f \cdot M$ by our general feasibility condition, we have that

$$S_t = \frac{W_0 - t_j \cdot M}{t_f - t_j} \leq \frac{t_f \cdot M - t_j \cdot M}{t_f - t_j} = M \ .$$

[6]The notion of a critical moment and the subsequent Lemma 5 are due to Cho et. al. [6].

So long as we fully utilize all processors when available, S_t can never exceed M, and, by Lemma 5, we can never end up with more than M tasks with zero laxity. □

The rules given in Theorem 3 are about as simple a set of criteria as one could hope for. In essence, "If a job needs to be started now in order to finish on time, then start it now. If a job is finished, then stop it now. Don't let processors sit idle." These three rules would be an obvious minimum requirement for any scheduler that hoped to achieve feasibility on a task set with 100% utilization. Yet, when we require proportional workloads be completed at all system deadlines, they are also *sufficient*. As they are so simple, they leave plenty of room to design scheduling algorithms that attempt to reduce the number of context switches and task migrations.

4.2. Stack-and-Slice (SNS) Scheduling

Perhaps the simplest possible scheduling based on our three rules is a "Stack-and-Slice" (SNS) approach, which is best described visually[7]. First, make a "block" of length u_i for each T_i, and stack these blocks up along a number line (in any order), starting at zero. Their total length will be M (still operating under the assumption of 100% utilization). Slice this stack of blocks into length one chunks at $1, 2, \ldots,$ and $M-1$, and assign each chunk to its own processor. Each length 1 chunk of tasks represents the scheduling of tasks on the respective processor; tasks which are sliced in half will migrate between their two processors. See Figure 6 for an illustration with 7 tasks and 3 processors. To find the actual timing points of context switches (at local job completion events), simply multiply each length 1 segment by the length of the current scheduling window.

It is clear from the description and the figure that Rules 1-3 of Theorem 3 are satisfied by this scheme: tasks are turned off when (locally) complete, the only tasks to achieve zero laxity are scheduled just in time, and each processor is always running a distinct task. Tasks which migrate are run at the beginning of the window on one processor, and at the end on the other. So long as such a task has utilization no more than 1 (which is required for *any* feasible schedule), its running times on the two processors will be disjoint. This gives the straightforward SNS scheduling algorithm: compute the context switch times indicated in the diagram (partial sums of task utilizations), reduce modulo 1 for each processor, and multiply times by the length of the current window. Except for this last multiplication, this can all be done once as a pre-processing step, so long as the task set is static. Note that there is *no* computational overhead at secondary events: here, a "scheduling event" (which in many algorithms requires iterating through all jobs, performing various calculations, or even sorting them) is merely following a pre-determined instruction to replace one task with another on one processor; no "decisions" are made.

[7] SNS is a simplification of the EKG [3] scheduler

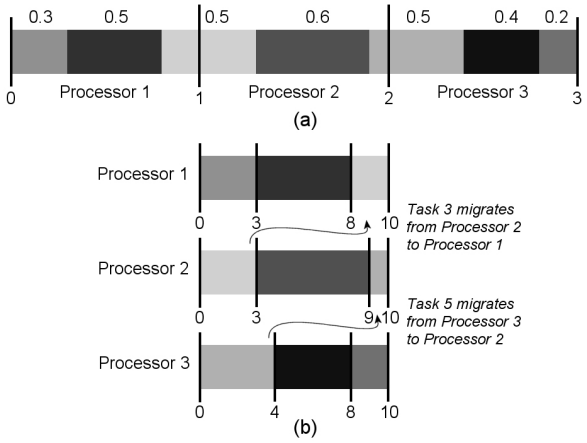

Figure 6. Stack-and-Slice Scheduling
(a) Seven tasks with utilizations shown above. These are lined up in arbitrary order, then sliced at length 1 intervals. **(b)** Each processor runs its task set over a length 10 time window. Jobs sliced in (a) are seen migrating in (b).

Notice that, in general, there will be $M-1$ tasks which are required to migrate. Further, if we repeat a predetermined stack-and-slice ordering for each time window, each of these $M-1$ tasks will migrate *twice* per window: once in the middle, and again at the end, when it moves back to its starting processor. We can cut this number of migrations in half simply by reversing the ordering of tasks on each processor in odd-numbered windows. Looking at the example in Figure 6, task 3 runs for the first 0.3 of the window on processor 2, then for the last 0.2 on processor 1. If we reverse the ordering within each processor for the next window, then task 3 will *start* on processor 1 (for 0.2) and then *finish* on processor 2 (for 0.3). As a result, tasks will only migrate in the middle of time windows, and never at the ends; we can limit the number of migrations to $M-1$ per window.

Theorem 6 *The stack-and-slice scheduling scheme with mirroring in odd windows will produce at most $N-1$ context switches and $M-1$ migrations per window.*

Proof. The $M-1$ migrations were discussed above. In the worst case, each job except the very first will cause a context switch when it is turned on. With mirroring, each processor will be running the same job at the end of one period and the beginning of the next, so no context switches occur at the end/beginning of scheduling windows. This results is $N-1$ context switches per window. □

5. Survey of Recent Sufficiently Fluid Algorithms

We now explain and analyze a number of recent papers in the context of our new framework. We start with the first two recent papers to utilize the notion of proportional deadlines to achieve optimality [3] [6], which appeared in the latter half of 2006. Several papers since then have borrowed approaches from one or the other, and tackle various sub-optimal variations of our scheduling problem. We will look at the two original papers in some detail, and then survey others that have followed.

5.1. Improvements with EKG

The EKG algorithm presents a modified Stack-and-Slice scheme. First, we observe that the $N - M + 1$ non-migrating tasks actually form a partitioned task set; that is, each one is permanently assigned to a single processor. The time between the migrating tasks at the beginning and end of a window consists of a single processor running jobs assigned exclusively to it. Instead of observing proportioned deadlines for these tasks, we may simply schedule them with EDF. In Figure 6, for example, we might view Processor 1 as a uniprocessor system with 80% capacity, and two tasks with utilizations of 0.3 and 0.5. In scheduling these two tasks, we essentially ignore all other tasks in the system, and run the simple, uniprocessor-optimal EDF algorithm. In this way, all tasks except the migrators may be "decoupled" from the proportional deadline scheme, allowing them to complete with fewer context switches. This method is known as *task splitting* or *semi-partitioning*.

The $M - 1$ migrating tasks, however, are still tightly coupled. The task split between Processors 1 and 2 must be scheduled in sync with the one on 2 and 3, which must be synced with the task split on 3 and 4, etc. However, the time windows need only be defined by the deadlines of the migrating tasks, and larger time windows translates to fewer total migrations. The EKG algorithm also allows for a partial decoupling of these migrating tasks, at the cost of optimality. The algorithm divides and decouples processors into groups of k, each of which operates independently of other groups. The extra space at the end of a group is not filled in with a partial task, but because each group is independent of the others, its scheduling windows are only based on its own $k - 1$ migrating tasks. With fewer and larger windows, fewer context switches are required in general. When $k = M$, we have optimality; when $k = 2$, we can guarantee only 66% utilization, but with significantly fewer context switches and migrations.

5.2. The T-L Plane Visualization

The *Time and Local Execution Time Plane* ("T-L Plane") model of Cho et. al. [6] provides a convenient means of visualizing deadline windows. The T-L Plane is represented as an isosceles right triangle, with time on the horizontal axis, and work remaining on the vertical axis. As time moves forward, a task's work remaining curve will not follow its fluid rate curve, but instead will be in one of two modes:

- If it is running, it will have a slope of -1, since both axes are on the same scale
- If it is idle, it will have a slope of 0

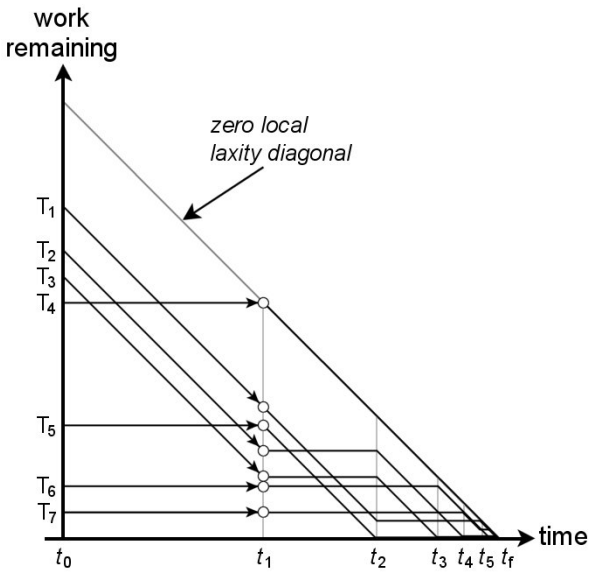

Figure 7. Work Remaining Curves in a T-L Plane
Seven tasks on three processors. At each secondary ZERO LAXITY or WORK COMPLETE event, the three jobs with least laxity are executed.

A task's work remaining curve will alternate between these two modes as it is turned on and off. (Figure 2 gives a simple picture of the progression of one job through this plane, drawn next to its fluid rate curve.) By time t_f, all curves must have height 0 (i.e., completed their work for this time window.) These are reset to heights proportional to their rates for the beginning of the next window. A ZERO LAXITY event is an inactive job (horizontal curve) hitting the hypotenuse; a WORK COMPLETE event is an active job (slope = -1) hitting the horizontal axis. A sample T-L Plane with executing jobs can be seen in Figure 7.

At t_0 and at each secondary event, the LLREF scheduler sorts the jobs by work remaining, and activates the M highest jobs. While this policy ensures successful completion of all tasks (it can easily be seen to guarantee the conditions of Theorem 3), it has a high computational overhead compared to EKG, and causes more context switches than are needed to satisfy Theorem 3. Also, this policy gives no prescription for the assignment of jobs to processors; it does not address the issue of task migration.

5.3. A Note on Performance

While SNS is designed primarily to be simple, it is instructive to compare its performance to the other algorithms. We compared SNS and LLREF on 1000 random task sets on 2-20 CPUs and found that SNS consistently generates about 1/3 as many context switches and migrations as LLREF[8]. Because EKG is similar to SNS but with optimizations to further limit task migrations, it can be expected to produce even fewer migrations, but at the cost of somewhat greater computational overhead.

5.4. Subsequent Work

A number of subsequent algorithms have expanded on ideas introduced with EKG and LLREF. They do not represent improvements in optimal scheduling, per se. Instead, they reduce context switches and migrations for sub-100% utilization task sets, and address variants of our basic scheduling problem. There are two main classifications: those that follow EKG and use task-splitting, and those that use variants on LLREF's T-L Plane model.

Andersson and Bletsas [1] propose an EKG variant for dealing with sporadic task sets (that is, where job arrival times are irregular and unknown in advance). Time windows are bounded by small, fixed width intervals rather than task deadlines. Like EKG, a tunable parameter (corresponding to window widths) is available, and can bring schedulable utilization arbitrarily close to 100% at the cost of more context switches and migrations. A sequel to this paper introduces the EDF-SS(DTMIN/δ) algorithm [2], which extends the problem to tasks with arbitrary deadlines (which may fall before, on, or after irregular arrival times). It also uses fixed-width time windows, but attempts to split tasks with the smallest minimum deadlines.

The Ehd2-SIP algorithm [12] is also similar to EKG, but sacrifices optimality for improved general performance. It starts with a general stack-and-slice task-to-processor assignment, but with tasks stacked in increasing period order. Then, rather than utilizing any notion of fluid scheduling, it uses EDF to schedule tasks on each processor, subject to the following exception: the "right-hand" half of a split task always has highest priority on its processor, unless it's "left-hand" other half is already running on the adjacent processor. This scheme only allows for 50% processor utilization in the worst case, but generally has high scheduling success until total utilization reaches the 80-90% range. These success rates are higher than strict partitioning EDF algorithms, although Ehd2-SIP suffers from more preemptions; conversely, EKG tends to have a higher success rate at the cost of more preemptions. The sequel to Ehd2-SIP is EDDP [13], which uses EKG's scheme of only partially filling processor capacity. It also schedules each processor with EDF, but removes priority for the right half of a split task, and artificially adjusts deadlines to improve schedulability. Its worst case utilization improves to 65%, but performance is otherwise similar.

The E-TNPA algorithm [11] extends the T-L Plane/ LLREF algorithm with two major improvements. First, when the total utilization is under 100%, it runs an excess time apportionment algorithm at the beginning of each window, and distributes unused CPU time among tasks in the form of increased workloads within the window. Once this is done for a new window, it runs as if it were a normal T-L Plane, but with different rates. In this way, E-TNPA is *work-conserving*, that is, it never idles a processor when any task has unfinished work in its current period. The second improvement is the realization that the sorting of tasks by laxity at each scheduling event within the window is unnecessary. Instead, the paper merely claims that tasks can instead be ordered based on the needs of the application/environment. This still requires scheduling invocations at every ZERO LAXITY and WORK COMPLETE secondary event. The authors provided a modified approach to their work-conserving scheduler with TRPA [10]. Instead of apportioning free time at the beginning of a window, they now allow tasks to run arbitrarily (subject to the zero laxity rule) until such time as remaining required work in the window to meet all fluid schedule goals equals the remaining processor capacity in the window. Both E-TPNA and TRPA mimic the T-L Plane/LLREF system when given 100% utilization task sets, and like LLREF, they only prescribe *which* tasks should be running at a given time. With no scheme for assigning tasks to processors, it is difficult to gauge the potential overhead of migrations.

6. Future Work

It is our hope that the insights we have provided into optimal hard real-time scheduling will encourage new directions in future research. The simplicity of conditions for sufficiently fluid schedules, and the SNS algorithm based on those conditions, leave room for exploration and improvement. In the SNS algorithm, no prescription is given for ordering the stacked events before they are sliced. While finding some "best" ordering may be NP-Hard, some simple ordering heuristics may lead to practical improvements. For example, in sub-100% utilization cases, EKG assigns each sufficiently large tasks to its own dedicated processor. It also isolates groups of processors, so that tasks on one group need only observe deadlines within their group. Other decoupling schemes may be possible to reduce the adherence of all tasks to all system deadlines.

On the theoretical front, there is still a considerable gap between the *necessity* of these conditions applied to general

[8] In actuality, as LLREF does not assign jobs to processors, we had to devise a greedy method for this in an attempt to minimize migrations.

jobs, and their *sufficiency* when applied to all tasks over-constrained by all system deadlines. We would like to see sufficient conditions for optimality which depended on a less strict set of deadline constraints.

There are also new directions for improvements in sub-optimal algorithms available by utilizing our notion of dual schedules. As shown by the example of Figure 4, the inclusion of zero-laxity events from the dual can improve the success of a standard LLF scheduler. Perhaps other suboptimal algorithms could benefit from this additional insight. Also, there is a tempting symmetry in the notion of dual-symmetric schedulers; we are currently working expanding this idea and testing several resultant algorithms.

There is much that could be done on relaxed versions of our basic scheduling problem. As several surveyed papers show, when utilization is sub-100%, there are numerous avenues for heuristic improvements. Another generalization of the model includes dynamic task sets. Adding or removing tasks from the SNS scheduler should be relatively easy, but careful ordering of tasks on the "stack" could make the process more efficient. Another generalization is the asymmetric processor environment, where different CPUs have different clock speeds. SNS easily accommodates this by simply adjusting the integer "slice" points to represent each processor's relative speed. Numerous other variations on the scheduling problem exist and are being explored.

We believe the most important open problem is implementation. Our "optimal" algorithms cannot achieve their theoretical best performance in practice, because they don't reserve time for context switches and migrations. Current scheduling algorithms still favor the strict partitioned approach due to these overheads. It is unclear whether the schedulability improvements realized by optimal schedules can compensate for their additional cost.

7. Conclusion

There have been a number of recent advances in scheduling algorithms for periodic task sets in hard real-time, multiprocessor environments. What was missing was a recognition of their shared traits and insights, and an underlying theory to explain their success. This paper provides such a theory. We started by examining the inherent problems of older, greedy approaches to multiprocessor scheduling, and introduced the model of scheduling duality to clarify these problems. We then presented and proved a simple set of sufficient conditions for a scheduling algorithm to achieve optimality by periodically matching its theoretic fluid rate targets. We also presented the very simple SNS scheduler based on these conditions. Finally, we described a number of recent algorithms in terms of this simplified framework, in order to better understand why they work and what they have in common. We hope that our model aids in understanding past work, and contributes to the direction of future research.

References

[1] B. Andersson and K. Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 243–252, 2008.

[2] B. Andersson, K. Bletsas, and S. Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors. *International Real-Time Systems Symposium (RTSS)*, pages 385–394, 2008.

[3] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 322–334, 2006.

[4] S. Baruah. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15(6):600–625, 1996.

[5] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on Scheduling Algorithms, Methods, and Models*, 2004.

[6] H. Cho, B. Ravindran, and E. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. *International Real-Time Systems Symposium (RTSS)*, pages 101–110, 2006.

[7] S.-K. Cho, S. Lee, A. Han, and K.-J. Lin. Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems. *IEICE Transactions on Communications*, E85-B(12):2859–2867, 2002.

[8] S. Dhall and C. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[9] O. Dong-Ik and T. Bakker. Utilization Bounds for N-Processor Rate MonotoneScheduling with Static Processor Assignment. *Real-Time Systems*, 15(2):183–192, 1998.

[10] K. Funaoka, S. Kato, and N. Yamasaki. New Abstraction for Optimal Real-Time Scheduling on Multiprocessors. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 357–364, 2008.

[11] K. Funaoka, S. Kato, and N. Yamasaki. Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, 2008.

[12] S. Kato and N. Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 441–450, 2007.

[13] S. Kato and N. Yamasaki. Portioned EDF-based Scheduling on Multiprocessors. *ACM International Conference on Embedded Software (EMSOFT)*, pages 139–148, 2008.

[14] J. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1):209–219, 1989.

[15] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[16] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[17] S. Oh and S. Yang. A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks. *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–36, 1998.