A Scheduling Algorithm and Utilization Bound for Uniform Multiprocessors

by

Archana Meka

(Under the direction of Shelby H. Funk)

Abstract

We present a new scheduling algorithm, U-LLREF, which is the first scheduling algorithm for that is optimal for uniform multiprocessors. It is an extension of the LLREF algorithm for identical multiprocessors. These algorithms attempt to emulate a fluid scheduling model, which executes all periodic tasks at a constant rate equal to their utilization. Like the LLREF algorithm, U-LLREF generates schedules based on the fairness notion. It uses the Time and Local execution time (TL) plane to describe fluid schedules and ensures the amount of work completed by each task remains close to the corresponding amount of work in the fluid schedule by monitoring task progress within TL planes. **Keywords: LLREF, optimal scheduling algorithm, uniform multiprocessors**

A Scheduling Algorithm and Utilization Bound for Uniform Multiprocessors

by

Archana Meka

B.E., Osmania University, 2006

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2008

A Scheduling Algorithm and Utilization Bound for Uniform Multiprocessors

by

Archana Meka

Approved:

Major Professor: Shelby H. Funk

Committee: John A. Miller
Robert W. Robinson

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2008

Dedication

To my advisor, mentor and guide Shelby H Funk, my parents and Suman

Acknowledgments

First and foremost, I would like to thank my advisor, Shelby H. Funk, for all her supervision, support and guidance from the very early stages of this research till the end. I would also like to thank my committee members, John A. Miller and Robert W. Robinson for their valuable time. I would also like to thank the Computer Science department at the University of Georgia and its staff for letting me use their library and resources to accomplish this research. Lastly, I want to thank my parents for giving me the opportunity to come to the United States and to study at the University of Georgia. Without this, none of this research would have even been possible. They have shown me lots of love and support during my time here during the tough times and the good times. I thank Suman for his support and encouragement even during tough times. I also thank my brother for giving me advice from his own personal experiences at graduate school.

Table of Contents

Page

Acknowledgments v

List of Figures vii

Chapter

1 Introduction 1

2 Model and Definitions 6

3 Related work 10

4 U-LLREF: The LLREF Algorithm for Uniform Multiprocessors 18

5 Conclusion 26

Bibliography 28

List of Figures

2.1 Events in a TL-plane 9

3.1 No multiprocessor online algorithm can be optimal. 11

3.2 The level algorithm 14

3.3 Processor sharing 14

Chapter 1

Introduction

A real-time system is defined as a system which achieves logical computations within deterministic time constraints. These systems may be divided into two categories – hard and soft. In a soft real-time system, jobs may miss some deadlines. In a hard real-time system, any missed deadline is considered a system failure. Therefore a real-time system needs to ensure that specific deadlines are met to avert such failures. Usually a real-time system has a limited amount of processing capability. In order to meet its requirements, the processing capacity of the system needs to be allocated in such a way that all the time deadlines are met properly. In the case of multiple events occurring within a short period of time, the system must schedule carefully in order to finish processing jobs within their time boundaries. The method the real-time system uses to determine job execution is known as a real-time scheduling policy.

Real-time Scheduling

Real-time scheduling works with the notion of priorities. When allocating system resources for scheduling tasks, job properties are used to determine their priorities. Jobs are usually characterized by their release time, computation time and deadline. Many real-time systems have jobs that repeat at regular intervals. These repeated jobs are generated by periodic tasks.

Tasks

There are several different types of tasks performed by real-time systems, including periodic and sporadic tasks. These tasks can be described by their time constraints and the times at which consecutive jobs are released.

A periodic task releases jobs regularly at fixed time intervals. If the deadline of each job is equal to the arrival time of the next job, we say that deadlines equal periods. This thesis considers only that type of task.

Jobs generated by sporadic tasks can arrive at irregular time intervals, but will have some known time bounds. The period of a sporadic task is the bounded rate at which jobs arrive (i.e., the minimum inter-arrival time).

The scheduling algorithm selected to execute a task set will depend on the different types of tasks in the set. Scheduling algorithms may be preemptive or non-preemptive. A preemptive algorithm will interrupt the currently executing job if another job with a higher priority becomes ready to execute. However, if a task is non-preemptive no job can be interrupted until it is completed. This thesis introduces U-LLREF, a preemptive scheduling algorithm.

Processors

Real-time systems may be executed on a variety of different types of processing platforms, including single processor systems and multiprocessor systems. In a single processor system (uniprocessor) there is exactly one shared processor available and all jobs executed are run on that processor. In the case of a multiprocessor system, there are several processors available for jobs to execute on. The number of processors in a multiprocessor system is an important factor in determining the algorithm for real-time task scheduling and multiprocessor scheduling algorithms tend to be more complex than those for uniprocessors. The focus of this thesis will be on multiprocessor systems.

Multiprocessor Scheduling

Multiprocessors have different classifications depending of the performance of the individual processors. Common classifications include identical multiprocessors, uniform multiprocessors and unrelated multiprocessors.

In identical multiprocessor systems, a job's execution rate is independent of the processor that is currently executing the job. In these platforms, each processor has the same processing power.

In uniform multiprocessors systems, a job's execution rate depends on which processor is currently executing the job. In these platforms, each processor has an associated speed s. If a job executes on a speed-s processor for t time units, the processor performs $s \times t$ units of work.

In unrelated multiprocessor systems, a job's execution rate depends both on processor that is currently executing the job, and on the job currently being executed. Each processor/job pair has an associated speed $s(p, j)$. These platforms describe systems with specialized processors, such as graphics coprocessors, or digital signal processors.

This thesis focusses on uniform multiprocessor platforms.

Scheduling Algorithms for MultiProcessors

Tasks are scheduled in real-time systems using some scheduling algorithm. Scheduling algorithms may be divided into two categories, depending on whether there is a single scheduler controlling all processors and tasks, or there are multiple schedulers – each controlling a single processor and a specific subset of the tasks.

Global Scheduling Algorithms: Algorithms with a single scheduler controlling all processors and tasks are called global scheduling algorithms. These algorithms store the tasks that have arrived but have not yet finished their execution in a single queue, which is shared among all processors. If the multiprocessor contains m processors, then at every moment at most m tasks in the queue are selected for execution. Global scheduling algorithms may allow preempted tasks to move to a different processor and resume execution. If so, we say the algorithm allows migration.

Partitioned Scheduling Algorithms: Algorithms in which each processor has its own local scheduler, which schedules a specified subset of the tasks, are called partitioned scheduling

algorithms. Under partitioning, tasks are not allowed to migrate, hence the multiprocessor may be viewed as many uniprocessor systems executing simultaneously.

We say an algorithm is optimal if it meets all deadlines whenever it is possible to do so. There are two global optimal scheduling algorithms for periodic tasks on identical multiprocessors – namely pFair [3, 1], and LLREF [6]. To date, no optimal scheduling algorithm exists for uniform multiprocessors. This paper introduces U-LLREF, an extension of LLREF for uniform multiprocessors and proves the algorithm is optimal.

Both Pfair and LLREF are based on the fluid scheduling model and the fairness notion. In the fluid scheduling model each task executes at a constant rate at all times. This model is impractical because it requires processors to execute multiple jobs simultaneously. The LLREF and Pfair algorithms try to mimic the fluid schedule. For both these algorithms, decisions regarding task execution on the multiprocessor ensure that the algorithm has performed the same amount of work as the fluid schedule at specific points in time $t_0, t_1, t_2, \ldots$.

The Pfair algorithm makes scheduling decisions with the goal of ensuring that the distance between the fluid schedule and the actual schedule is always less than 1. The algorithm divides tasks into subtasks, called pseudo-tasks. Each pseudo-task executes for one time unit. Pseudo-tasks are given deadlines that guarantee the bound on the distance between the fluid and actual schedules. The pseudo-tasks are assigned priority according to their deadlines, with earlier deadlines having higher priority. If two pseudo-tasks have the same deadline, there are additional rules used for breaking ties.

The LLREF algorithm makes all scheduling decisions within the time and location remaining execution-time planes (TL planes); these planes describe the system behavior during any interval $[t_i, t_{i+1}]$. The schedule is divided into TL planes of various sizes. Therefore, the feasibility of the entire system depends on the schedule within each TL plane. This paper presents an optimal scheduling algorithm for periodic tasks executing on uniform multiprocessors. Our new scheduling algorithm, U-LLREF, extends LLREF to apply to uniform multiprocessors.

The remainder of this thesis is organized as follows. Chapter 2 introduces our model, defines all terms, and discusses LLREF in more detail. Chapter 3 introduces results that are relevant to this research. Chapter 4 introduces the U-LLREF scheduling algorithm and proves it is correct. Finally, Chapter 5 provides concluding remarks.

Chapter 2

Model and Definitions

Real-time systems are comprised of a processing platform and a set of *jobs*, $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$. A job is a sequential piece of code that executes on a processor. Jobs are described using the 3-tuple (a_i, c_i, d_i), where a_i, c_i and d_i are job J_i's arrival time, execution requirement and deadline, respectively. The execution requirement, c_i, is the amount of time J_i must execute to complete its work on a speed-1 processor. A schedule is correct if and only if J_i completes c_i units of work during the interval $[a_i, d_i)$. In this work, we assume all jobs are independent.

Real-time jobs are often generated by *periodic tasks*. A periodic task $T_i = (p_i, c_i)$ generates jobs $T_{i,k}$ at times $k \cdot p_i$ for $k = 0, 1, 2, \ldots$. Job $T_{i,k}$'s deadline is the arrival time of task T_i's next job, namely $(k+1) \cdot p_i$. One important task parameter is its *utilization*, $u_i = c_i/p_i$, which measures the proportion of time a task must execute on a unit speed processor (i.e., a processor with speed $s = 1$). The utilization is the rate at which the fluid schedule executes each task. The work presented in this thesis considers the scheduling of *periodic task sets* [11], $\tau = \{T_1, T_2, \ldots, T_n\}$. The total utilization and maximum utilization of task set τ are denoted U_{sum} and u_{sum}, respectively.

$$\begin{aligned} U_{sum} &= \sum_{i=1}^{n} u_i \\ u_{sum} &= \max_{1 \le i \le n} \{u_i\} \end{aligned}$$

We will be introducing an algorithm for executing periodic task sets on *uniform multiprocessors*, in which each processor has an associated speed. We denote an m-processor uniform multiprocessor $\pi = [s_1, s_2, \ldots, s_m]$. In a mild abuse of notation, we let s_j denote both the processor itself and its speed. A job executing on processor s_i for t time units performs $s_i \cdot t$

units of work. The total speed of π, denoted $S(\pi)$, is the sum of all the processor speeds, $S(\pi) = \sum_{i=1}^{m} s_i$. This is the maximum amount of work that can be performed on π in one unit of time.

A task set τ is said to be *feasible* on a multiprocessor π if there exists some way of scheduling all jobs to meet their deadlines. A scheduling algorithm is *optimal* if it can successfully schedule every feasible task set to meet all deadlines.

This research extends the LLREF [6] scheduling algorithm which currently applies only to identical multiprocessors. As stated in Chapter 1, this algorithm tracks the fluid schedule by ensuring that LLREF and the ideal schedule have completed the same amount of work at certain points in time. For all times $t \geq 0$, the ideal schedule has completed $u_i \cdot t$ units of work on task T_i at time t. By contrast, the LLREF algorithm is guaranteed to complete $u_i \cdot t$ units of work whenever *any* task has a deadline (i.e., at any time $k \cdot p_i$ for some integer $k \geq 0$ and some $i = 1, 2, \ldots, n$). We denote these points in time $t_{f,0}, t_{f,1}, \ldots$.

At each time t_e such that $t_{f,k-1} \leq t_e < t_{f,k}$ for some $k > 0$, LLREF makes all scheduling decisions based only on the *local remaining execution requirement* and the *local utilization* for each task T_i, denoted $\ell_{i,e}$ and $r_{i,e}$, respectively. The local execution requirement is the amount of work that must be completed on task T_i between time t_e and time $t_{f,k}$ in order for LLREF to have performed $u_i \cdot t_{f,k}$ units of work by time $t_{f,k}$. The local utilization is the ratio of local execution requirement and the remaining amount of time,

$$r_{i,e} = \frac{\ell_{i,e}}{t_{f,k} - t_e}.$$

The *total local utilization* at time t_e, denoted R_e, is the sum of all tasks' local utilization.

$$R_e = \sum_{i=1}^{n} r_{i,e}.$$

The Time and Local execution requirement plane (or TL-plane) is a two dimensional plane, whose horizonal axis is time (t) and vertical axis is local remaining execution requirement (ℓ). Each TL-plane illustrates task behavior between two deadlines $t_{f,k-1}$ and $t_{f,k}$. At the beginning of each TL-plane, each task's local utilization is initialized to the task's global

utilization. Hence, the execution done by the LLREF algorithm will match the execution done by the fluid schedule at all deadlines.

The status of each task is represented using a *token*. If task T_i is executing on processor s_j at time t, then T_i's token moves down in the plane with a slope of of $-s_j$. If T_i is not executing, T_i's token moves horizontally. LLREF's scheduling decisions are designed to ensure the following:

- No task executes more than its ideal amount within the TL-plane – i.e., $\ell_{i,t} \geq 0$ for all $t \geq 0$ and $i = 1, 2, \ldots, n$.
- No remaining execution requirement gets so large that it cannot be completed before the next deadline $t_{f,k}$.

These two constraints have two corresponding events, which are illustrated in Figure 2.1. The first event, corresponding to $\ell_{j,t}$ reaching a value of 0 is called a *bottom* (or B) event. In the TL-plane, a B event occurs when a task's token reaches the bottom of the TL-plane. The second event, corresponding to a task (or tasks) needing to execute continually for the remaining time in order for it (them) to meet its (their) deadline, is called a *ceiling* (or C) event. If only one task has had a C-event, then that task's token will intersect the line with a slope of $-s_1$ that intersects the vertical axis at $t_{f,k}$ — if the task does not execute on the fastest processor for the remaining time, it will certainly not track the ideal schedule at the next deadline, $t_{f,k}$.

On identical processors, there is only a single type of C event condition, namely when a task's local utilization reaches 1. As we will see in the next section, the presence of different processor speeds on uniform multiprocessors results in different types of C events — if one task has already had a C event requiring it to only execute on s_1, then the next task's C event will not be able to execute on s_1! Instead, tasks on uniform multiprocessors have C_k events, which occur when k tasks have 0 laxity on the k fastest processors.

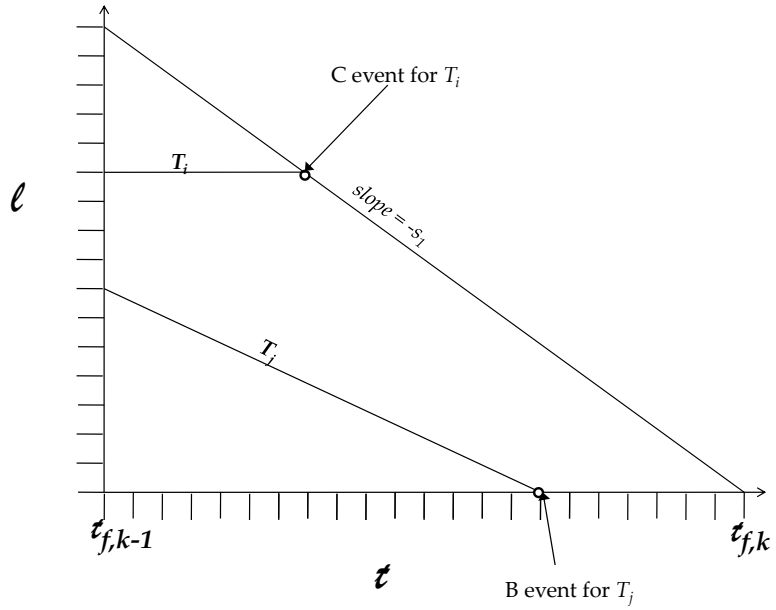


Figure 2.1: Events in a TL-plane

Chapter 3

Related work

This chapter presents some results on the work previously done on uniform multiprocessors. First we look at the results of Hong and Leung, then the results of Dertouzos and Mok.

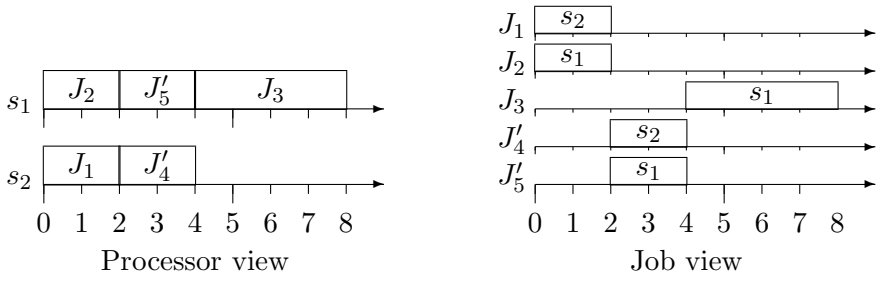
Two independent groups, Hong and Leung [9] and Dertouzos and Mok [7], proved that there can be no optimal online algorithm for scheduling real-time instances on identical multiprocessors. On the positive side, Baruah, *et al.* [3] developed a job-level dynamic-priority scheduling algorithm called Pfair, which they proved is optimal for periodic tasks on multiprocessors. Srinivasan and Anderson [13] later proved that the Pfair algorithm could also be used for sporadic task sets with a bit of modification.

Theorem 1 ([9]) *No optimal online scheduler can exist for instances with two or more distinct deadlines for any m-processor identical multiprocessor, where* $m > 1$.

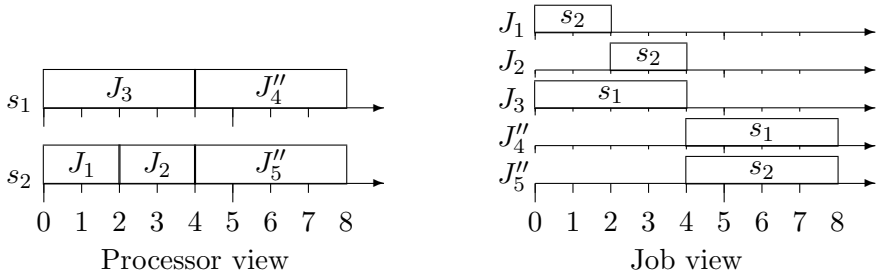
This theorem was proved by Hong and Leung by the following counterexample.

Example 1 ([9]) Consider executing instances on a two-processor identical multiprocessor. Let $I = \{J_1 = J_2 = (0, 2, 4), J_3 = (0, 4, 8)\}$. Construct I' and I'' by adding jobs to I with later arrival times as follows: $I' = I \cup \{J'_4 = J'_5 = (2, 2, 4)\}$ and $I'' = I \cup \{J''_4 = J''_5 = (4, 4, 8)\}$. There are two possibilities depending on the behavior of J_3.

Case 1: J_3 **executes during the interval [0,2).** Then one of the jobs of I' will miss a deadline. Inset (a) of Figure 3.1 [8] illustrates a valid schedule of I' on two unit-speed processors. Notice that the processors execute the jobs J_1, J_2, J'_4 and J'_5 and never idle during the interval $[0, 4)$. Moreover, these four jobs all have the same deadline at $t = 4$.



(a) Job J_3 cannot execute in interval $[0, 2)$



(b) Job J_3 must execute in interval $[0, 2)$

Figure 3.1: No multiprocessor online algorithm can be optimal.

Therefore, if J_3 were to execute for any time at all during this interval, it would cause at least one of the jobs to miss its deadline.

Case 2: J_3 **does not execute during the interval [0,2).** Then one of the jobs of I'' will miss a deadline. Inset (b) of Figure 3.1 illustrates a valid schedule of I'' on two unit-speed processors. Notice that the processors execute the jobs J_1, J_2, and J_3 during the interval $[0, 4)$ and all three jobs have completed execution by time $t = 4$. Moreover, jobs J_4'' and J_5'' both require four units of processing time in the interval $[4, 8)$. If job J_3 did not execute during the entire interval $[0, 2)$, it would not complete execution by time $t = 4$. Therefore, it would require processing time in the interval $[4, 8)$ and at least one of the jobs J_3, J_4'', or J_5'' would miss its deadline.

Therefore, the jobs in I cannot be scheduled in a way that ensures valid schedules for all feasible job sets without knowledge of jobs that will arrive at or after time $t = 2$.

Dertouzos and Mok [7] also considered online scheduling algorithms on identical multiprocessors and determined that to optimally schedule jobs, there must be three job properties that must be known.

Theorem 2 ([7]) *For two or more processors, no real-time scheduling algorithm can be optimal without complete knowledge of 1) the deadlines, 2) the execution requirements, and 3) the start times of the jobs.*

They also found conditions in which if one or more of these job properties are not known, it is still possible to determine a valid schedule. They determined that, in general real-time systems, it is possible to schedule jobs that don't necessarily arrive simultaneously without knowing their arrival times as long as it is possible to schedule jobs when they do arrive simultaneously.

Horvath, *et al.* [10], studied the scheduling problem of a set of non-real-time preemptable jobs arriving simultaneously. They showed that the minimum makespan for independent non-real-time jobs is based on the total execution requirements of the jobs as well as the total speeds of the processors. The makespan of a system is the amount of time required to complete all the jobs in the system. The following theorem is integral to the development of U-LLREF.

Theorem 3 *Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of n independent non-real-time jobs, each with arrival time equal to zero and indexed according to non-increasing execution requirements, $c_i \geq c_{i+1}$ for all $i = 1, 2, \ldots n-1$, and let C_i denote the cumulative execution requirement of the i largest jobs for all $i = 1, 2, \ldots, n$, and let $S_i(\pi)$ denote the cumulative speed of π's i fastest processors. Then for any uniform heterogeneous multiprocessor, π, the minimum makespan for scheduling I on π is*

$$\omega = \max\left(\max_{1 \leq i \leq m}\left\{\frac{C_i}{S_i(\pi)}\right\}, \frac{C_n}{S(\pi)}\right) \ . \tag{3.1}$$

Horvath, *et al.* [10], introduced an algorithm called the *level algorithm* which always completes job execution at time ω. This algorithm assigns jobs to processors in a recursive manner. At time $t = 0$, this algorithm sets j equal to m and k equal to the number of jobs with the largest execution requirement or in other words, the jobs with the highest initial *level*. The job *level*, J, at time t is the amount of remaining work. If $k > j$, the k jobs are jointly assigned to the j processors, otherwise the k jobs are assigned to the k fastest processors and the remaining jobs are assigned to the slower processors respectively. This processor assignment will remain until either some set of jobs completes executing or the level of one group equals the level of the group below it.

Example 2 [8] Let $\pi = [5, 3, 1]$ and let I be the set of jobs with execution requirements $c_1 = 20, c_2 = 16, c_3 = 6$, and $c_4 = 5$. Figure 3.2 illustrates the level-algorithm schedule of I on π. Notice that the diagrams only have one time line for all the processors (or jobs) to more easily reflect when jobs execute jointly on one or more processors. Initially, all the jobs have distinct levels so J_1, J_2 and J_3 execute on s_1, s_2 and s_3, respectively. At $t = 1$, the levels of J_3 and J_4 are both equal to 5, so these two jobs jointly execute on s_3. At $t = 2$, the levels of J_1 and J_2 are both equal to 10, so these jobs jointly execute on processors s_1 and s_2. At $t = 3\frac{4}{7}$, all the jobs have a level of $3\frac{5}{7}$ so they all execute jointly on all three processors until they complete at $t = 4\frac{17}{21}$.

Jobs are scheduled and jointly assigned to processors in a round-robin fashion, where the shared intervals are divided into smaller intervals. For instance, if there are k processors, $s_1, s_2, \ldots, s_k$, and j jobs, $J_1, J_2, \ldots, J_j$, scheduled for time t where $k \leq j$, then the interval is into j subintervals of length (t/j). Each job then executes on each processor for exactly one subinterval and idles for $(k - j)$ subintervals. At the first subinterval, J_i executes on s_i where $1 \leq i \leq k$; following this, each job J_i executes on s_{i+1} where $1 \leq i \leq k - 1$. Figure 3.3 illustrates a schedule of five time units where five jobs share four processors [8].

The level of each job J_i not only depends on c_i if $\mathcal{J}$ has precedence constraints, but the level also includes the longest *chain* that starts at J_i. A chain is defined as a sequence of jobs

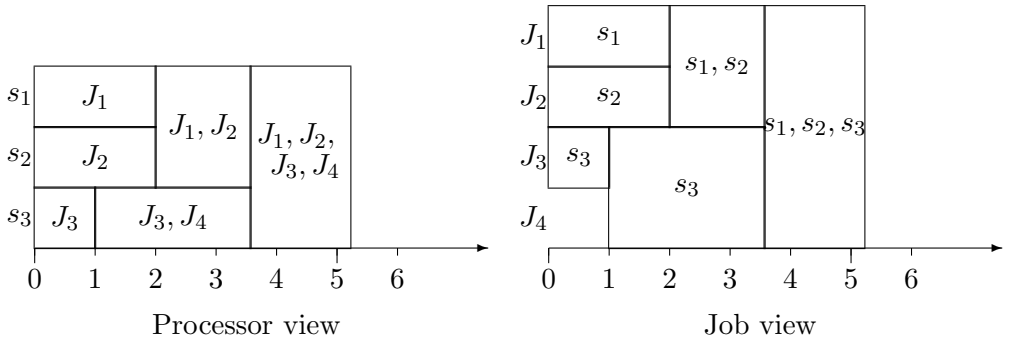


Figure 3.2: The level algorithm.

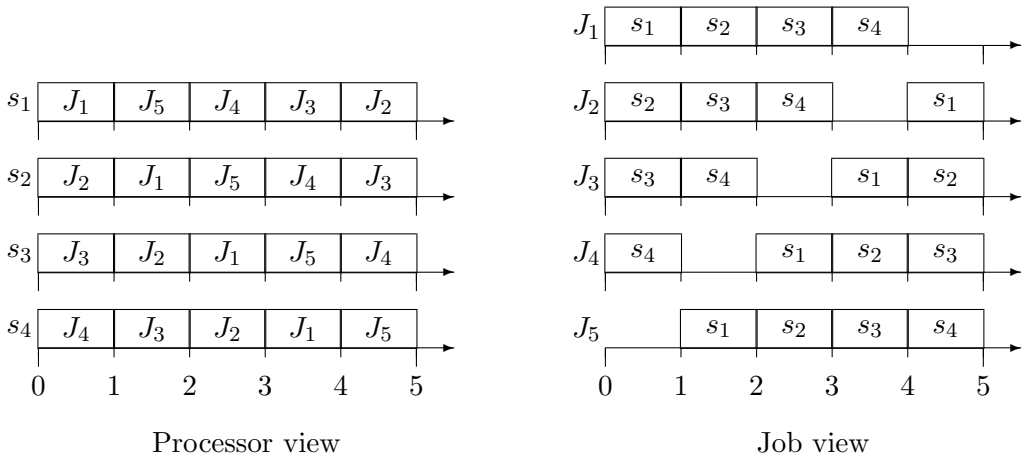


Figure 3.3: Processor Sharing.

$J_{i_1}, J_{i_2}, \ldots, J_{i_{n'}}$ such that $J_{i_k} < J_{i_{k+1}}$ for each $k = 1, 2, \ldots n' - 1$. The length of the chain is sum of its component jobs' execution requirements: $c_{i_1} + c_{i_2} + \ldots + c_{i_{n'}}$. Horvath, et al. [10], discovered that while this level algorithm may minimize the makespan for independent jobs, it does not minimize the makespan when $\mathcal{J}$ has precedence constraints.

3.0.1 Real-time scheduling on uniform heterogeneous multiprocessors

Baruah and Goossens developed an RM-schedulability test for uniform heterogeneous multiprocessors. The following section will describe the results from their test. **Rate monotonic scheduling.** Rate monotonic (RM) scheduling is a fixed-priority scheduling algorithm which assigns higher priority to tasks with smaller periods. Therefore, all jobs generated by a certain task have the same priority. The schedulability test developed by Baruah and Goossens [4, 5] tests the rate monotonic (RM) scheduling algorithm with full migration on uniform heterogeneous multiprocessors, as shown below.

Theorem 4 ([4, 5]) *Let τ be any task set and let π be any uniform heterogeneous multiprocessor. If*

$$U_{sum}(\tau) \leq \frac{1}{2}\big(S(\pi) - (1 + \lambda(\pi)) \cdot u_{max}(\tau)\big) \ ,$$

where $S(\pi)$ is the total speed of π and $\lambda(\pi)$ is π's identicalness parameter, then τ can be successfully scheduled on π using RM *with full migration.*

The difference between EDF and RM is that for EDF the active jobs gain higher priority as their deadline approaches. On the other hand, for RM, the jobs always have the same priority regardless of their deadlines. Because of this the utilization bound is smaller for RM than for EDF. This decreased bound compensates for the difficulties inherent in fixed-priority scheduling. In the remainder of this section we will discuss the results developed by Baruah concerning the dynamic scheduling algorithm EDF-F on uniform heterogeneous multiprocessors.

The robustness of EDF-F. Remember that π dominates π' if (1) $m(\pi) \geq m(\pi')$, and (2)$s_i(\pi) \geq s_i(\pi')$ for all i with $1 \leq i \leq m(\pi')$. Whenever I is A–schedulable on some

multiprocessor, if a scheduling algorithm A is guaranteed to successfully schedule any real-time instance I on π, then A is said to be *robust* with respect to domination. Baruah [2] showed that EDF-F is robust with respect to domination. This robustness is preferred because it allows processors to be upgraded without complete re-analysis of the entire system.

3.0.2 Comparison between Pfair, LLREF and Heterogeneous Partitioning

In this section we will discuss three different scheduling algorithms for multiprocessors and their strengths and drawbacks. The algorithms included in our discussion are Pfair, LLREF and Heterogeneous partitioning. Each of these algorithms is used for different applications in multiprocessor systems. Pfair and LLREF apply only to identical multiprocessors. Heterogeneours partitioning applies to unrelated multiprocessors. We begin with Pfair

One of Pfair's strengths is that it tries to reduce the amount of memory to L2 traffic which increases the latency in the normal processor operation. The Pfair algorithm can be used to schedule periodic and sporadic tasks, it is a quantum based scheduler, *i.e.*, tasks are allocated fairly in proportion to their weights. This algorithm assigns processor time in discrete quanta. One limitation of this algorithm is that it has no support for multi-threads.

The LLREF algorithm is slightly different and is more effective than the Pfair-based scheduler. The LLREF algorithm proposes a real-time scheduler for multi-processor environments and works only for periodic and independent tasks i.e., tasks that do not share resources. In this algorithm processors may be idle even when tasks are present in the ready queue. Tasks also execute at a constant rate at all times. This algorithm constantly tracks the allocated task execution rate through task utilization. Task execution behavior on multiprocessors is given by the time and local remaining execution time plane (TL plane) which was introduced in Chapter 2.

Both of these algorithms have a high overhead cost. In general, multiprocessor scheduling algorithms with low overhead also have low processor utilization. It is not uncommon for hard real-time systems to have 50% processor utilization. Pfair and LLREF both have very high

utilization. The theoretical utilization of both these algorithms is 100%. However, accounting for the high overhead for the scheduling algorithms will reduce the processor utilization to some degree.

The Heterogeneous Partitioning algorithm analyzes the mapping of n tasks to a set of m different types of processors. The Task partitioning algorithm tries to solve the partitioning problem in heterogeneous multiprocessors – i.e., it states the possible ways to map the set of n tasks onto a set of m processing units, where each processing unit is of a different kind. Jobs must be independent to use this algorithm – e.g., they may not share any exclusive resources. An Integer Linear Program (ILP) formulation is used to make the complex and intractable partitioning simple. The ILP is optimized using a routine LPRelax, which finds feasible mappings for feasible systems. The main drawback of this algorithm is that, instead of discussing the scheduling problem it focuses mainly on mapping sets of tasks to sets of processors.

Chapter 4

U-LLREF: The LLREF Algorithm for Uniform Multiprocessors

Cho, et al. [6], proved that LLREF is optimal on identical multiprocessors. If it is possible to schedule a periodic task set to meet all its deadlines on a m processor identical multiprocessor, then the LLREF schedule will meet all deadlines, *i.e.*, a task set satisfies $U_{sum} \leq m$ and $u_{sum} \leq 1$. Clearly, if $u_{sum} > 1$, then the task with the highest utilization will require more processor time than is available on a single processor even if the task executes 100% of the time. As a task can only execute on one processor at a time, having utilization greater than 1 would have to cause the task's deadline to be missed. Similarly, if $U_{sum} > m$, then the task set will require a larger proportion of processing than is available on m processors. Thus, the feasibility condition for LLREF proves that it is optimal on identical multiprocessors.

Because uniform multiprocessors have processors executing at different speeds, the test for feasibility on uniform multiprocessors is more complicated. Given a uniform multiprocessor $\pi = [s_1, ..., s_m]$ where $s_1 \geq s_2 ... \geq s_m$, and a task set $\tau = \{T_1, ..., T_n\}$ where $u_1 \geq u_2 \geq ... \geq u_n$, τ is feasible on π whenever the following conditions hold (we assume $n \geq m$) [10].

$$
\begin{aligned}
u_1 &\leq s_1, \\
u_1 + u_2 &\leq s_1 + s_2, \ldots \\
\sum_{i=1}^{k} u_i &\leq \sum_{i=1}^{k} s_i \text{ for } k = 1, 2, \ldots, m-1, \text{ and} \qquad (4.1) \\
U_{sum} &\leq S(\pi). \qquad (4.2)
\end{aligned}
$$

As described in Chapter 2, LLREF has critical and bottom scheduling events. For simplicity, at each event, the tasks are re-indexed in decreasing order by local utilization requirement – i.e., $r_{i,e} \geq r_{i+1,e}$ for $i = 1, 2, \ldots, n-1$. On identical multiprocessors, the m tasks with

highest local utilization are scheduled to execute at each event. On uniform multiprocessors, we have the additional requirement that task T_1 (which has the maximum local utilization after re-indexing) executes on processor s_1, task T_2 executes on processor s_2, etc. In general, task T_i executes on processor s_i for $1 \leq i \leq m$, and tasks $T_{m+1}, T_{m+2}, \ldots, T_n$ do not execute during the interval $[t_e, t_{e+1}]$.

4.0.3 Conditions for B and C Events

Recall that in LLREF, the scheduler is invoked in the midst of a TL-plane under two conditions. Either some task(s) complete executing, in which case we have a bottom (or B) event, or some task(s) laxity becomes zero, in which case we have a critical (or C) event. Below, we present conditions that must hold when these two types of events occur, starting with C events. Once these conditions have been found, we will show that if we reschedule whenever one of these events occurs, then the utilization condition given in 4.2 will always hold – i.e., the U-LLREF schedule will schedule the tasks without any deadline misses.

By Constraint 4.2 shown above, there are m different utilization bounds for scheduling on uniform processors. The first $m-1$ bounds require that the total local utilization of tasks T_1 through T_k is no more than the total speed of the k fastest processors. Each of these bounds has a corresponding *laxity constraint*, C_k in which k tasks have 0 laxity on the k fastest processors. If these k tasks do not execute on the k fastest processors for the remaining time in the TL-plane, at least one of them will not complete its local execution requirement by time t_f.

Whenever C_k holds at time t_c, the following condition holds for the k tasks with maximum local utilization.

$$\sum_{i=1}^{k} r_{i,e} = \sum_{i=1}^{k} S_i.$$

At a given scheduling event t_{e-1}, the next scheduling event will occur either when one of the $m-1$ critical conditions occurs or a bottom condition occurs. Below we discuss how to determine which of these conditions will cause the next event.

Lemma 1 *Let $T_{i_i}, ..., T_{i_k}$ be tasks that cause a C_k event at time t_c. Thus T_{i_j} is assigned to processor s_{i_j} at time t_{e-1}, the previous scheduling event*[1]*. Then $t_c = t_{e-1} + \Delta t$, where*

$$\Delta t = \frac{\sum_{j=1}^{k}(r_{i_j,e-1} - s_j)}{\sum_{j=1}^{k}(s_{i_j} - s_j)}.(t_f - t_{e-1}).$$

PROOF If these tasks cause a C_k event at time $t_c = t_{e-1} + \Delta t$ then it must be the case that

$$\sum_{j=1}^{k} r_{j,c} = \sum_{j=1}^{k} s_j.$$

Also, by definition of $r_{j,c}$, we know that

$$\sum_{j=1}^{k} r_{j,c} = \sum_{j=1}^{k} \frac{\ell_{j,c}}{t_f - t_c}.$$

Hence,

$$\sum_{j=1}^{k} s_j = \sum_{j=1}^{k} \frac{l_{j,c}}{t_f - t_c}. \tag{4.3}$$

During the interval $[t_{e-1}, t_c]$, task T_{i_j} completes $\Delta t \cdot s_{i_j}$ units of work (for ease of notation, assume $s_{i_j} = 0$ if $i_j > m$). Therefore, $l_{j,c} = l_{j,e-1} - \Delta t \cdot s_{i_j}$. Substituting this into Equation 4.3 gives

$$\begin{aligned}
\sum_{j=1}^{k} s_j &= \sum_{j=1}^{k} \left(\frac{\ell_{i_j,e-1} - \Delta t \cdot s_{i_j}}{t_f - t_c} \right) \\
\Rightarrow \sum_{j=1}^{k} s_j &= \sum_{j=1}^{k} \left(\frac{\ell_{i_j,e-1} - \Delta t \cdot s_{i_j}}{t_f - (t_{c-1} + \Delta t)} \right) \\
\Rightarrow (t_f - (t_{c-1} + \Delta t)) \cdot \sum_{j=1}^{k} s_j &= \sum_{j=1}^{k} \left(\ell_{i_j,e-1} - \Delta t \cdot s_{i_j} \right).
\end{aligned}$$

[1]Note that task T_{i_j} becomes task T_j after re-indexing at time t_c

Calculating for Δt gives

$$\begin{aligned}\Delta t &= \frac{\sum_{j=1}^{k}(\ell_{i_j,e-1} - (t_f - t_{e-1}) \cdot s_j)}{\sum_{j=1}^{k}(s_{i_j} - s_j)} \\ &= (t_f - t_{e-1})\frac{\sum_{j=1}^{k}(r_{i_j,e-1} - \cdot s_j)}{\sum_{j=1}^{k}(s_{i_j} - s_j)}.\end{aligned}$$

The last step follows from the definition of $r_{i_j,e-1}$.

To date, we have not found an efficient method for determining the k tasks that will cause the earliest C_k event. For each $k = 1, 2, \ldots, m-1$, we have a conjecture that the k tasks that cause an earliest C_k event are a subset of the $(k+1)$ tasks that causes an earliest C_{k+1} event. If we are able to prove that this conjecture is true, then we can efficiently find all the C_k events by repeatedly finding the next task to add to the set causing each of the C events.

Of course, a C event will only occur if a B event does not occur first. Next we determine the condition that leads to a B event.

A B event occurs when some task completes its local remaining execution. When this occurs, the scheduler should allow some other task to execute rather then allowing the processor to stay idle. Therefore, we need to identify when a B event will occur so we can invoke the scheduler to reschedule the tasks at the appropriate time.

Event B occurs when there exists some $i \leq m$ such that token T_i has no local remaining execution time. While on identical multiprocessors, T_m is the only task that can invoke a B event, on uniform multiprocessors any of the executing tasks can invoke a scheduling event. This is because all processors execute at the same speed on identical multiprocessors, whereas on uniform multiprocessors it is possible for some processor s_k with $k < m$ to execute so

much more quickly than s_m that $T_{k,e}$ finishes before $T_{m,e}$ even though $\ell_{k,e} > \ell_{m,e}$. The lemma below shows how to determine when a B event occurs.

Lemma 2 *Assume a B event occurs at time t_b and that t_{e-1} is the scheduling event that precedes the B event. Let $T_{b,e-1}$ be the task that is scheduled to execute at time t_{e-1}. Then $\ell_{b,e-1}/s_b \leq \ell_{j,e-1}/s_j forall j = i, \ldots, m$.*

PROOF Let t_b be the time when a B even occurs. For $k = 1, \ldots, m$, each task $T_{k,e-1}$ executes at speed s_k during the interval $[t_{e-1}, t_b]$. Because T_b causes a B event, it performs $\ell_{b,e-1}$ units of execution during the interval $[t_{e-1}, t_b]$ – i.e., $t_b = \ell_{i,e-1}/S_i$.

Assume that for some task T_a, the lemma does not hold. Then, T_a does not cause a B event before T_b and $\ell_{b,e-1}/s_b \leq \ell_{a,e-1}/s_a$. If T_a executes on processor s_a, it will complete its execution at time $t_a = \ell_{a,e-1}/s_a < t_b$, which contradicts our assumption.

This completes the proof.

4.0.4 FEASIBILITY TEST

We have seen the conditions under which B and C events occur. We will now show that if a task set τ adheres to the utilization conditions above (conditions 4.2), then U-LLREF will successfully schedule τ on π. We begin by demonstrating necessary and sufficient conditions (similar to conditions 4.2) for a task to miss one of its local deadlines, t_f. We then show that if the tasks are rescheduled whenever a B or C event occurs then all tasks will meet their local deadlines within each TL-plane. Hence, U-LLREF will successfully schedule any feasible task τ set on the uniform multiprocessor τ.

Theorem 5 *Assume τ is any task set scheduled on some uniform multiprocessor π using the U-LLREF scheduling algorithm. Then a task of τ will miss the k^{th} local deadline, $t_{f,k}$, if and*

only if one of the following conditions is violated at some time t_e, where $t_{f,k-1} \leq t_e < t_{f,k}$.

$$
\begin{aligned}
r_{1,e} &\leq s_1, \\
r_{1,e} + r_{1,e} &\leq s_1 + s_2, \ldots \\
\sum_{i=1}^{k} r_{1,e} &\leq \sum_{i=1}^{k} s_i \text{ for } k = 1, 2, \ldots, m-1, \text{ and} \qquad (4.4) \\
R_e &\leq S(\pi). \qquad (4.5)
\end{aligned}
$$

PROOF Assume a task T_i misses the local deadline $t_{f,k}$. Then $\ell_{i,f_k} > 0$. Therefore, there exists some $\epsilon > 0$ such that at time $t_\epsilon = t_{f,k} - \epsilon$, we have $\ell_{i,\epsilon} > s_1 \cdot epsilon$. Hence, the first condition above is violated.

Now assume that one of the above conditions is violated at some time t_e. Then either $R_e > S(\pi)$ or there exists some k such that $\sum_{i=1}^{k} r_{i,e} > \sum_{i=1}^{k} s_i$. In the first cases, the total local execution requirements of all tasks is greater than $(t_{f,k} - t_e)S(\pi)$. By definition, the maximum amount of work that can be done on π during the interval $[t_e, t_{f,k}]$ is $(t_{f,k} - t_e)S(\pi)$. Therefore, π will not be able to perform the required amount of work before the local deadline $t_{f,k}$. In the second case, we have k tasks whose total local execution requirement exceeds $(t_{f,k} - t_e) \sum_{i=1}^{k} s_i$. Therefore, if these tasks are to meet their local deadlines they will need to execute on more than k processors. This can only happen if some task executes on two processors simultaneously, which is not permitted. Hence, once again π will not be able to perform the required amount of work before the local deadline $t_{f,k}$.

We now show that if τ satisfies conditions 4.2 on π and the tasks are rescheduled whenever a B or C_k event occurs, then condition 4.5 will always be satisfied.

Theorem 6 *Let τ be a set of n tasks and let π be an m-processor uniform multiprocessor. Assume condition 4.5 holds at some time t_{e-1} and let $t_e = \min t_{c_k}, t_b$ be the next B or C_k event that occurs. Then condition 4.5 will hold at time t_e.*

PROOF By definition of a C_k event, we know that the first $(m-1)$ conditions will continue to hold – namely, $\sum_{i=1}^{k} r_{i,e} \leq \sum_{i=1}^{k} s_i$. Hence, it suffices to prove that $R_e \leq S(\pi)$.

By assumption, we know that

$$R_{e-1} \leq S(\pi). \tag{4.6}$$

Define α as follows.

$$\alpha = \frac{t_f - t_{e-1}}{t_f - t_e} = (1 + \frac{t_e - t_{e-1}}{t_f - t_e}).$$

Multiplying both sides of equation 4.6 by α gives

$$\begin{aligned} \alpha R_{e-1} &\leq \alpha S(\pi) \\ \Rightarrow \alpha R_{e-1} &\leq (1 + \frac{t_e - t_{e-1}}{t_f - t_e}) S(\pi) \\ \Rightarrow \alpha R_{e-1} - (\frac{t_e - t_{e-1}}{t_f - t_e}) S(\pi) &\leq S(\pi) \end{aligned} \tag{4.7}$$

For $i = 1, 2, \ldots, n$, assume that when the tasks are re-indexed at time t_e, task T_i becomes task T_{i_e}. Let $\Delta t = t_e - t_{e-1}$. By definition, task T_i executes on processor s_i during the interval $[t_{e-1}, t_e]$. Hence, for all $i = 1, 2, \ldots, n$,

$$\ell_{i_e,e} = \ell_{i,e-1} - \Delta_t s_i. \tag{4.8}$$

(As in Lemma 1, assume $s_i = 0$ if $i > m$.)

By definition of α, we have

$$\begin{aligned} \alpha &= \frac{t_f - t_{e-1}}{t_f - t_e} \\ \Rightarrow t_f - t_e &= \frac{t_f - t_{e-1}}{\alpha} \end{aligned} \tag{4.9}$$

Combining this with the definition of $r_{i_e,e}$, we know that

$$\begin{aligned} r_{i_e,e} &= \frac{\ell_{i_e,e}}{t_f - t_e} \\ \Rightarrow r_{i_e,e} &= \frac{\ell_{i_e,e}}{\frac{t_f - t_{e-1}}{\alpha}} \\ &= \alpha \frac{\ell_{i_e,e}}{t_f - t_{e-1}} \end{aligned} \tag{4.10}$$

Substituting Equation 4.8 into Equation 4.10 gives

$$\begin{aligned} r_{i_e,e} &= \alpha \frac{\ell_{i,e-1} - \Delta_t s_i}{t_f - t_{e-1}} \\ &= \alpha \left(r_{i,e-1} - \frac{\Delta_t s_i}{t_f - t_{e-1}} \right) \\ \Rightarrow r_{i_e,e} &= \alpha \left(r_{i,e-1} - (\frac{t_e - t_{e-1}}{t_f - t_{e-1}}) s_i \right). \end{aligned}$$

Taking the total of the local execution over all tasks and noting that $m \leq n$ implies that all the processors are executing tasks during the interval $[t_{e-1}, t_e]$ gives

$$\begin{aligned} R_e &\leq \alpha \left(R_{e-1} - (\frac{t_e - t_{e-1}}{t_f - t_{e-1}}) S(\pi) \right) \\ &= \alpha R_{e-1} - (\frac{t_e - t_{e-1}}{t_f - t_e}) S(\pi). \end{aligned}$$

In the first step, we have an inequality rather than an equality because if $m > n$ then some processors may be idling and the total speed of all *active* processors may be less than $S(\pi)$. However, if $m \geq n$, then all processors will be busy for the entire interval $[t_{e-1}, t_e]$ – if a processor idles, then a B event will occur. The second step above follows by the definition of α.

Substituting this into Equation 4.7 gives $R_e \leq S(\pi)$.

Corollary 1 *The U-LLREF algorithm is optimal on uniform multiprocessors.*

Proof This follows directly from Theorem 6.

Chapter 5

Conclusion

This paper introduces U-LLREF, the extension of the LLREF algorithm for uniform multiprocessors, an optimal scheduling algorithm for uniform multiprocessors. This is the first optimal algorithm for executing periodic tasks on uniform multiprocessors.

U-LLREF is designed to closely emulate a fluid schedule, in which all tasks execute at a constant rate. The algorithm makes all scheduling decisions locally within the TL-planes, and the amount of work performed by each task within TL-plane exactly matches the amount of work performed by the fluid schedule over the same period of time. U-LLREF operates by identifying events when tasks should be rescheduled – namely, B events, when a task completes its local execution requirement, and C_k events, when a set of k tasks must execute on the k fastest processors in order to perform their full workload within the TL-plane. Currently, we have a conjecture that would allow us to efficiently identify these k tasks. In the future, we hope to prove this conjecture is true, thereby making U-LLREF a polynomial time algorithm.

We would also like to extend U-LLREF to be defined for more general models such as sporadic tasks [12, 7], in which the period is indicated by the minimum amount of time between consecutive jobs (as opposed to the exact amount of time assumed in the periodic task model). We would also like to extend U-LLREF to handle non-independent tasks, such as tasks with resource constraints or precedence constraints.

Finally, we would like to extend the algorithm to reduce some of the overhead due to preemptions and migrations. Currently, we are exploring the possibility of swapping the execution requirements of jobs between TL-planes. For example, if tasks T_i and T_j swap

some execution requirement, then at the end of the TL-plane, T_i would be ahead of its fluid schedule and T_j would be behind its fluid schedule. In subsequent TL-planes, the tasks could swap back again so they both match their fluid schedules before their deadlines. This would reduce the number of tasks active in a TL-plane, which in turn would reduce the overhead due to B and C events. Of course, any such swap would have to be done carefully in order to insure that no deadlines are missed. Other methods of reducing overheads that we are considering include allowing tasks with higher local utilization to execute on slower processors in order to avoid migration costs.

Bibliography

[1] Sanjoy K. Baruah. Strong P-fairness: A scheduling strategy for real-time applications. In *Proceedings of the IEEE Workshop on Real-Time Applications*, 1994.

[2] Sanjoy K. Baruah. Robustness results concerning EDF scheduling upon uniform multiprocessors. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 95–102, Vienna, Austria, June 2002. IEEE Computer Society Press.

[3] Sanjoy K. Baruah, Neil Cohen, C. Greg Plaxton, and Don Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[4] Sanjoy K. Baruah and Joël Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, 2003.

[5] Sanjoy K. Baruah and Joël Goossens. Rate-monotonic scheduling on uniform multiprocessors. In *Proceeding of the 23rd International Conference on Distributed Computing Systems*, Providence, RI, April 2003. IEEE Computer Society Press.

[6] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors.

[7] Michael Dertouzos and Aloysius K. Mok. Multiprocessor scheduling in a hard real-time environment. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.

[8] Shelby Funk. *EDF Scheduling on Heterogeneous Multiprocessors.* PhD thesis, CS Department, UNC Chapel Hill, 2004.

[9] Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 244–250, Huntsville, Alabama, December 1988. IEEE.

[10] Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, 1977.

[11] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[12] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[13] Anand Srinivasan and James Anderson. Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*. Scheduled for publication.