Power Efficient Scheduling for Hard Real-Time Systems on a Multiprocessor Platform

Peter J. Nistler, Jean-Luc Gaudiot

Power Efficient Scheduling for Hard Real-Time Systems on a Multiprocessor Platform

Peter J. Nistler, Jean-Luc Gaudiot

University of California, Irvine
Irvine, CA, USA
peter.nistler@gmail.com, gaudiot@uci.edu

Abstract. An online, real-time scheduler is proposed to minimize the power consumption of a task set during execution on a multiprocessor platform. The scheduler is capable of handling the spectrum of task types (periodic, sporadic, and aperiodic) as well as supporting mutually exclusive, shared resources.

The solution presented is a user adjustable scheduler which ranges from producing an optimal schedule which requires the minimum power during the worst case execution scenario to producing a suboptimal schedule which aggressively minimizes power during the typical execution scenario.
Keywords: Real-Time, Scheduler, Power Management, Multiprocessor

1 Introduction

Multi-core systems are designed to bring the processor power consumption and heat density down to a manageable level without sacrificing performance. This claim is based on the dynamic power characterization of a CMOS device: $P = C * V^2 * f$ (1), where P is power, C is the device capacitance, V is the voltage, and f is the frequency. At the same time, we are entering a world of ubiquitous computing.: there are microprocessors in watches, phones, televisions, simple kitchen appliances, cars, and power grids. Many of these systems need to operate within certain time bounds to ensure proper function. Embedded systems are often designed within a prescribed power envelope. This power constraint might be defined by thermal boundaries or by energy constraints, such as in a battery or solar cell-operated environment.. Performance and power requirements must be carefully weighed against each other to ensure that the system functions as desired. With the current trend towards multiprocessors and the need to minimize power, there is a demand for a hard real-time multiprocessor scheduler that optimizes for minimal power usage.

2 Prior Art

Much work has dealt with creating power efficient, real-time schedulers, with one of two goals: minimize the power for the worst case scenario. or for the typical scenario.. The first type of scheduler, *(e.g.,* Chen et al. [1]) assumes that the task will always

require the maximum possible execution time possible. Under this assumption, the scheduler scales back the operating frequency of the processor so that the task completes exactly at its deadline. This is an optimal solution for scheduling a given task set on a multiprocessor while providing hard real-time guarantees. The second type of scheduler, (*e.g.,* Gruian [2] and Malani *et al.* [3]), assumes that the task will execute in an average time and sets out to minimize the power. The scheduler sets the operating frequency such that the task on average will finish execution by a set time $t_a < t_{deadline}$. If the task does not finish executing by t_a then the scheduler increases the operating frequency such that the task can finish the task execution by $t_{deadline}$. Fig**. 1** depicts the differences in the speed scaling schemes.

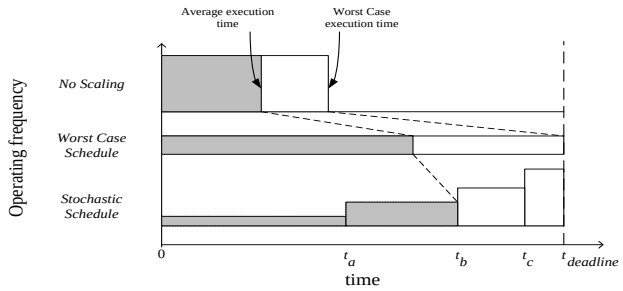

Fig. 1. Scheduling Strategies

Finally, Chen and Hsueh [4] introduced a novel framework for executing periodic real-time tasks on uniform multiprocessors. They developed an optimal, simple, clear, and easily visualized scheduler.. Their framework is a core component of this work. The Precaution Cut Greedy (PCG) scheduling algorithm is *fluid* in that each task executes at a constant rate during its executable period. It is also *P-fair* in that the work allocation and work executed are within one quantum at all times.The basic quantum employed by the PCG algorithm is a novel construct called a *Time and Local Execution Requirement* (T-L_{er}) plane. The T-L_{er} plane can be visualized as a right isosceles triangle where the y-axis is the system execution requirement (work) and the x-axis is time. The plane is created by overlaying each task's fluid schedule inside the triangle and aligning the time boundaries with scheduling events. By assuming the period of every task is equal to its deadline, the only scheduling events in Chen and Hsueh's model are the task periods. The entire system schedule is broken into a series of T-L_{er} planes. Inside each plane, the tasks can be executed in any fashion, provided that all the required work is completed by the end of the plane. The PCG algorithm provides an optimal solution for scheduling on uniform multiprocessors, through the application of the fluid scheduling model.

3 Problem Definition

The existing power efficient real-time schedulers are not well suited for use in their current form. If a scheduler can only handle task sets where all the tasks share one common deadline; if it can only handle periodic tasks; if it cannot deal with even the

simplest of locks or other shared resources; then the scheduler has limited applicability to actual implementations. The focus of this work is to produce that well-rounded scheduler, a scheduler designed with an actual system in mind.

The first part of this work establishes the hardware and power model used in the system. The second part describes the basic scheduler and supporting framework that minimizes the power consumption on a multiprocessor. This approach is strictly for hard real-time systems. The system will contain support for any mix of periodic, sporadic, and aperiodic tasks. The support for these non-periodic tasks requires that the scheduler and framework are online algorithms. The scheduler shall also maintain a strict ordering of accesses to shared resources such as memory mapped IO structures and locks. The result is an online scheduler and supporting framework that minimizes the power consumption of a multiprocessor while scheduling hard real-time tasks.

The third part of this work explains how to use the scheduler and supporting framework to support the two different approaches to scheduling. The first approach describes a high-reliability system. The system is simple, clean, and guarantees an optimal solution for the worst case execution scenario. It is a combination and further extension of the work done by [1] and [4]. The second approach describes a more commercial solution, optimized for the typical execution scenario. It combines the high-reliability approach with the idea of stochastic scheduling.

The work is broken up into three different sections. Section 4 discusses what the power optimal configuration of the hardware looks like. Section 5 extends Chen and Hsueh's scheduling framework for supporting periodic, sporadic, and aperiodic tasks while supporting shared resources. Section 6 describes how to tailor the scheduler so that it can proactively reduce the system power.

4 Power Optimal Hardware Configuration

This section addresses the problem of selecting processor operating frequencies so that the system can support the requirements of a given task set. The power optimal hardware configuration and a simple algorithm for determining it are provided.

4.1 The Processor / Task Relationship

A relationship needs to be established between the processor capabilities and the task execution requirements. First a multiprocessor model is selected to establish a foundation for the work. Based on this model's relationship between the software and hardware some bounds can be established to find the minimum operating frequencies required to support the task set.

There are two different high-level multiprocessor models that need to be explained before the real work can begin. The first model is identical parallel machines. Each of the processors in the system is identical to all the other processors and operates at the same frequency. Thus all the processors in the system perform identically, allowing tasks to freely migrate without affecting the task's execution. This model is the easiest to support at the operating system and scheduler level.

The second model is uniform parallel machines. Similar to the identical parallel machines the tasks can freely migrate as an identical instruction set used on all the processors. The difference is that the processing capabilities s_j of the processors are only linearly related to each other. The linear relation is $s_j = e/t$ where e is the amount of work executed and t is the time elapsed. Each processor in this system is allowed to take on any positive value for s_j. Regardless of the scheduler's goals it must be aware of how the processors compare to each other to make a good decision.

The multiprocessor model being used in this research is a special case of the uniform parallel machine model. The processors are physically implemented as identical parallel machines. However, each of the processors can independently operate at a unique clock frequency. There are several reasons for taking this approach. The first reason is that the software designer only needs to optimize the code to execute on a single processor microarchitecture. The second reason is that this approach relates processing capability and operating frequency to a simple, linear formula. If the processing capability is normalized to the maximum operating frequency, then it follows that $f_j = s_j * f_{max}$. The final reason is that running separate clock frequencies to each core allows a fine-tuning of the power consumption.

As described earlier the amount of work (machine instructions) that a processor can complete in time t is defined by: $e = s_j * t$. The tasks being scheduled are described by the triplet: $T_i(c_i, d_i, p_i)$ where task T_i has a worst case execution time c_i at the maximum frequency f_{max}, a relative deadline d_i, and a period p_i. Note that c_i is a measure of time, rather than a measure of work. It follows that the work required to execute T_i must be $e_i = s_{max} * c_i$ where s_{max} is the processing capability of a processor at its maximum frequency. Recall with that $s_{max} = 1$, the relationship simplifies to $e_i = c_i$. The work of a task equals its worst case execution time.

A common metric for determining schedule feasibility in real-time systems is task utilization of a processor u_i. The utilization on a fixed frequency, uniprocessor is described as $u_i = c_i/d_i$. This equation needs a slight update in order for it to work on a variable frequency system; the actual execution time t must meet the equality $t \leq d_i$. When the utilization is combined with the equation for s_j it follows that

$$u_i = \frac{c_i}{d_i} = \frac{e_i}{t} = s_j \ \ when\ t = d_i.$$

This is to say that a processor must have a processing capability $s_j \geq u_i$ in order for the task to finish by its deadline. Further, it becomes apparent that the minimum operating frequency of the processor executing T_i is simply $f_j = u_i * f_{max}$. This equation is a lower bound on the selectable frequencies that can be used while executing a given task. This utilization model can be extended to the system level, where it can be seen that the total utilization of the system is characterized by

$$U = \frac{1}{m}\sum_{k=1}^{n} u_k \quad \textbf{(2)}$$

where m is the number of processors and n is the number of tasks in the system. As a quick check for task set feasibility, note that if, at any point in time either $U > 1$ or $u_i > 1$ then the task set can never be successfully scheduled on the system.

4.2 The Ideal Processor Frequencies

This section shall investigate the optimal processor speeds for a given workload, without regard for the real-time constraints. The only goal of this section is to determine the optimal operating frequency of each processor such that a given amount of work can be accomplished in a set amount of time.

As mentioned earlier, the power of a system is quadratic relative to the operating voltage of the device and linear in terms of f. Furthermore, the relationship between the required operating voltage and the frequency on dynamic voltage scaling hardware is assumed to be a linear relationship ($V = k_1 * f + k_2$). Since V is linear with f the result is that $P \propto C * f^3$ when frequency and voltage are scaled together.

The power model can be easily extended to support multiprocessor systems. By summing the power of each individual processor, the system's power is modeled by

$$P \propto C * \sum_{k=1}^{m} f_k^3 \quad \textbf{(3)}$$

This is not simply the power of one processor multiplied by m, because each of the processors can take on a unique frequency. From the earlier equality $s_j = u_i$ and equation (2), it can be seen that a system must maintain a total processing capability $S = \Sigma_{k=1}^{m} s_k = U * m$ to ensure that all the tasks in a given span will complete by their deadlines. If $S > U * m$, then the system is capable of processing more work than the task set demands. On the other hand, if $S < U * m$ then the system cannot process all the task set demands, which means that the task set is infeasible by the system.

Theorem 1. *The system power to achieve a total processing capability* $\boldsymbol{S}$ *is minimized when* $\boldsymbol{f_j = \frac{S}{m} * f_{max}, 1 \leq j \leq m}$.

Case 1: $m = 1$

This case describes a uniprocessor system. All the work must be executed on the one processor.

$$f_1 = S * f_{max}$$

Case 2: $m = 2$

In this case the total processing capacity $S = s_1 + s_2$, which leads to the system power characterization of

$$P \propto C * f_{max}^3 * (s_1^3 + s_2^3) \; where \; s_2 = S - s_1.$$

This system functions correctly as long as the original assumption $0 \leq s_j \leq 1$ holds true. The result is that the minimum power is achieved when the work is evenly distributed across the processors.

$$f_1 = f_2 = \frac{S}{2} * f_{max}$$

Case 3: $m > 2$

The optimal solution for this case can be established with a proof by contradiction. Assume that there is a system where the optimal power is achieved by unevenly

distributing the work among all the processors. This assumption can be expressed by the formula

$$\left(\frac{S}{m}-\epsilon\right)^3+\left(\frac{S}{m}+\frac{\epsilon}{k}\right)^3*k+\left(\frac{S}{m}\right)^3*(m-k-1)\leq\left(\frac{S}{m}\right)^3*m$$

Where $1\leq k\leq m-1 \;\; and \;\; -\frac{kS}{m}\leq\epsilon\leq\frac{S}{m}$.

The first equation states that there exists a state where one processor deviates from the average execution speed S/m by ϵ while k other processors adjust their frequency to ensure that $S=U*m$. The deviation ϵ must be bounded to ensure that none of the processors are set to a negative frequency. Solving the first equation for ϵ gives the result

$$\epsilon\geq\frac{3kS}{m(k-1)}.$$

When this equation is evaluated across the bounds of k it follows that $\epsilon>3S/m$. This result violates the bound on ϵ; one of the processors must run at a negative frequency in order for the first equation to hold true. This is nonsense. Therefore by contradiction, the minimum system power is achieved at

$$f_j=f_{avg}=\frac{S}{m}*f_{max},\qquad 1\leq j\leq m. \tag{4}$$

The result is that the hardware consumes the least power when the processors all operate at f_{avg}. This is to say the optimal solution from the hardware's perspective is to adjust each processor such that $s_j=U$. Therefore, the power optimal scheduler must try to produce a schedule where all the tasks can be feasibly scheduled with this average operating frequency.

4.3 Power Optimal Software Configuration

This section shall address the problem of selecting processor operating frequencies such that the system consumes the minimum power while maintaining task set feasibility. The system must find a solution that is as close to the hardware optimal solution as possible while producing a feasible configuration. This section will establish the requirements for the scheduling algorithm.

The job of selecting optimal processor frequencies can be reduced to arranging all the task utilizations into bins such that the total value of each bin is as close to U as possible. It is important to note that tasks in the system are assumed to be non-divisible. One task cannot be split into parallel threads and simultaneously executed on multiple processors to shorten the execution time. The task is only allowed to execute on one processor at a time. However the task is allowed to migrate between any processors without penalty. The feasibility constraint for uniform multiprocessors has already been described by Funk *et al.* [5]; their result follows.

Theorem 2 *Feasibility Condition on Uniform Multiprocessors* by Funk et al. [5] Consider a set $\tau = \{T_1, T_2, \dots, T_n\}$ of tasks indexed with non-increasing utilization (i.e., $u_i \geq u_{i+1}$ for all i). Let $U_i = \frac{1}{m}\Sigma_{j=1}^{i} u_j$ for all i. Let π denote a system of $m \leq n$ uniform processors with processing capabilities $s_1, s_2, \dots, s_m$ where $s_i \geq s_{i+1}$ for all i. Let $S_i = \Sigma_{j=1}^{i} s_j$ for all i. Task set τ can be scheduled to meet all deadlines on uniform multiprocessor platform π if and only if the following constraints hold: $U_n * m \leq S_m$ and $U_k * m \leq S_k$, for all $k = 1, \dots, m$

The speed scaling algorithm needs to select processor frequencies as close to f_{avg} as possible while meeting the constraints of Theorem 2. The first and easiest constraint is to ensure that only $m \leq n$ processors are provided to the scheduler. This is accomplished by idling or powering down any excess processors. Next the algorithm must ensure that the list of tasks is sorted by $u_i \geq u_{i+1}$. If the most demanding task in the set u_1 is less than or equal to U, then it is possible to schedule all the tasks when the active processors operate at f_{avg}. The solution for this case meets the constraints of Theorem 2 and is optimal from a power perspective.

When the value u_1 is greater than U a slightly different approach needs to be taken. In order to ensure that Theorem 2 is upheld set $s_1 = u_1$. In this situation one of the processors in the system must operate at $s_i \geq u_1$ in order to meet the constraints. Further, if P_1 runs at $s_1 = u_1$ then it is operating at the minimum frequency possible for meeting the deadline of T_1. In this situation the deviation ϵ from the necessary frequency for maintaining task set feasibility has been reduced to zero. This processor is running at the optimal frequency for the current condition. Once the frequency of P_1 has been thus set it can be ignored for the remainder of the speed scaling algorithm. Likewise, since the work required by T_1 has been accounted for, this task is ignored as well. The speed scaling algorithm is then recursively called on the reduced set of processors and tasks, where $U_{new} = \frac{1}{m-1}(m * U_{old} - u_1)$. By induction, this speed scaling algorithm produces a minimum power system configuration achieving task set feasibility. This speed scaling algorithm will be implemented as a piece of the LTF-M algorithm developed by Chen et al. in [1].

5 Building the Scheduler

The scheduler being proposed is composed of three components: the framework, the speed scaling algorithm, and a T-L_{er} plane scheduling algorithm. The framework is the mechanism which breaks the timeline into a series of T-L_{er} planes. It defines the boundaries of each plane and keeps track of which task needs to be executed in the plane. The framework provides the current task set to the speed scaling and T-L_{er} plane scheduling algorithms. The speed scaling algorithm, in turn, sets the operating frequency of each processor such that the system consumes the least power while maintaining a feasible hardware configuration. The task set, the length of the current plane, and the processor configuration are finally passed to the plane scheduling algorithm. The plane scheduling algorithm then performs its role of executing the tasks to meet the deadline on the current hardware configuration.

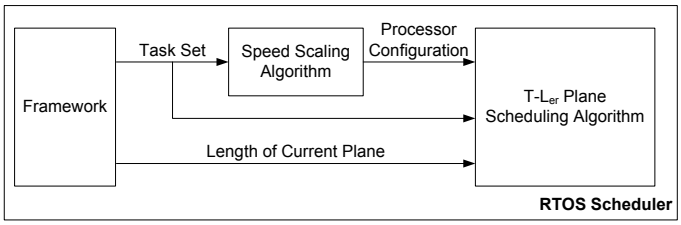

Fig. 2. Breakdown of the Scheduler

The framework based on Chen and Hsueh is overly constrained. It can only execute a subset of periodic tasks, it does not dynamically adjust to reduce power when a task completes early, it assumes that the scheduler can run in zero time, and it cannot handle shared resources, major hindrances to an actual implementation. This section describes the changes needed to allay these shortcomings.

5.1 Supporting More Task Types

The first shortcoming of the framework proposed by Chen and Hsueh in [4] is that it only supports a highly restricted set of task types. The authors assumed that the period of every task equals its deadline, $d_i = p_i$. This assumption is mutually exclusive with the ability to support sporadic and aperiodic tasks.

The solution is to extend the framework so that it supports task sets where $d_i \leq p_i$. The plane scheduler requires that every task's period and deadline line up with a T-L_{er} plane boundary. The framework will set the plane boundaries accordingly. The downside of allowing the deadline and period to take on different values is that there may be more T-L_{er} planes within the hyperperiod. Naturally, more T-L_{er} planes directly correspond to more invocations of the scheduler.

A sporadic task can be viewed as a specific type of periodic task. It has unique values c and d, an arrival time a, and $p = \infty$. However the sporadic task cannot be injected directly into the current task set or deadline violations might ensue. This is because the current T-L_{er} plane has already been configured for the existing task set. Either the processor frequencies would require adjustment within the plane, or the sporadic task must wait to begin execution until the subsequent plane. The framework will wait for the subsequent plane. There are two reasons driving this decision. The first reason is the high execution cost required to adjust the schedule mid-plane. The second reason is to minimize the number of changes to the processor frequencies. .This decision to wait can result in a large number of rejected sporadic tasks which might have been executable, if only they began executing earlier. This rejection rate can be reduced by having the framework insert extra plane boundaries into the system, thereby reducing the average length of the plane. The trade-off is an increased number of scheduler invocations.

The sporadic task acceptance test for this scheduler is based on Theorem 2. If this rule holds true, while including

$$u_{sporadic\ task} = \frac{c}{(a+d) - time_{first\ plane\ it\ could\ execute\ in}}$$

in τ, for all T-Ler planes in which the sporadic task executes (from the first boundary encountered after the sporadic task arrived, through the task's deadline), then it can be accepted. With this the scheduler is capable of handling sporadic tasks while providing real-time guarantees.

In a similar manner aperiodic tasks can be executed whenever the system has free resources. This means that provided $U_k * m < S_k$, for all $k = 1, \dots, m$ remains true, the scheduler can assign the aperiodic task a utilization of $u_{aperiodic\ task} \leq 1 - u_k$ and schedule the task within the current T-L_{er} plane.

5.2 Exploiting Hidden Speed-ups

The second shortcoming in Chen and Hsueh's work is that the work allocation and execution is assumed to be P-fair. In order to enforce this P-fair model, each task would need to be heavily instrumented to monitor its progress.

The solution is to use a weaker model which states that at all times the actual work executed is greater than or equal to the actual work allocation. This model gives the flexibility to run a task faster than the worst case execution time. This is important for implementation on a real system as there are many hidden factors which can improve the execution time. Several common examples are branch predictors, caches, and break statements nested inside of a loop. The weaker model allows the system to exploit these hidden speed-ups and does not require instrumentation of the code.

This weaker fairness model is only used in the framework. It cannot be driven into the scheduling algorithm without risking a significant increase in complexity. When a task finishes its execution it informs the framework that it has completed. The framework will remove the task from the task set until its next period, if applicable. The system utilization for the new plane is: $U_{new\ plane} = U_{original\ plane} - \frac{u_i}{m}$ where u_i is the utilization of the completed task.

Inside of a given T-L_{er} plane the scheduling algorithm will not be aware that a task is executing faster than worst case. The tasks will be scheduled onto the processors according to their utilizations which at this point is dictated by the worst case execution time. The result is that the schedule for the plane can be determined at the beginning of the plane. This pre-determined schedule will be followed for the duration of the plane without any further invocations to the scheduling algorithm..

The framework can further minimize the system power by inserting extra plane boundaries into the hyperperiod. The extra plane boundaries reduce the delay from when the task finishes to when the hardware configuration will be adjusted, at the next invocation of the speed scaling algorithm. As the time between boundaries approaches zero, the tasks are removed as soon as they finish. This means that the scheduler never requests more work from the system than is absolutely necessary. The price for this power savings is the overhead of calling the scheduler more frequently. This price is not as dramatic as it first seems if the framework inserts boundaries to break up a larger T-L_{er} plane into identical length smaller T-L_{er} planes. In the case where no task finishes early the schedule produced for each plane will be identical. The task set and t_f remain unchanged which guarantees the same result from the speed scaling and scheduling algorithm. The framework should recognize this and use the same schedule, saving significant execution time.

5.3 A Lightweight Scheduling Algorithm

Recall that the speed scaling and scheduling algorithms will be invoked prior to the start of every T-L_{er} plane. The large execution complexity will become unwieldy as the number of planes increase. The LTF-M algorithm developed by Chen et al. in [1] is a lightweight algorithm which optimally sets the processor frequencies and schedules tasks, provided that all the tasks share a common start time and a common deadline. These requirements are met at the level of the T-L_{er} plane. The LTF-M algorithm only has a complexity $O(n)$ yet is still optimal both from the feasibility perspective and from a power perspective.

5.4 Shared Resources

The next improvement for the scheduler is sharing resources. Currently the scheduler requires that every task executes independently with no ordering between them. However, a resource often allows only one task to access it at a time. This section shall explain how the scheduler framework supports mutually exclusive, shared resources and introduces time based locks to enforce the ordering between tasks.

Support for shared resources will be handled through mutex locks. These locks require a minor modification to the scheduler framework and do not violate any real-time guarantees. The change is most clearly explained with the help of Fig. 3.

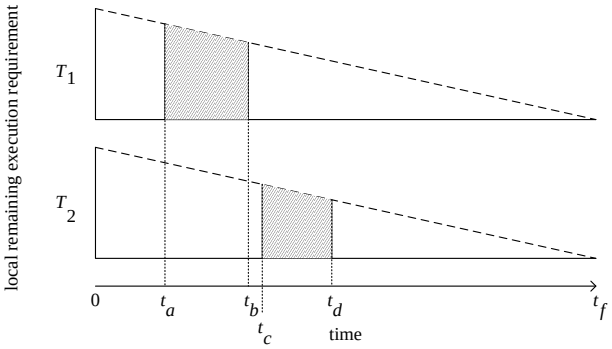

Fig. 3. Shared Resources in the Fluid Schedule

The system is executing two tasks, T_1 and T_2, where the shaded regions represent the time at which the shared resource is held by each task. The location and execution time of these critical sections must be provided to the framework. The framework will need to know the last possible moment at which the critical section may begin, t_a and t_c, and the last possible moment that the critical section may be exited, t_b and t_d. There will be no conflicts provided that $t_c \geq t_b$ or $t_d \leq t_a$ while the tasks execute according to their fluid schedules.

However, within a T-L_{er} plane each task does not necessarily execute according to its fluid schedule. For example, T_1 may not begin execution in a given plane until $t > 0$. This delayed start may result in a critical section finish time $t_b^{'} > t_c$. In this case T_2 would enter its critical section at t_c, violating the deadline guarantee.

The simple solution is to force the tasks to acquire and release the resource at fixed points in time: e lock must be requested and released at the granularity of a whole T-L_{er} plane. This is accomplished by having the framework insert plane boundaries at the requesting edge of a critical section. Provided that $t_b \leq t_c$ in the original fluid schedule, setting only t_c as a boundary ensures that $t_b^{'} \leq t_c$.

As the framework prepares to pass the upcoming plane's task set and deadline to the LTF-M algorithm it will perform a quick check. If there are multiple calls to a common lock, then the framework will insert a boundary at the beginning of the second request. Effectively the current plane will be shortened to run from the current time to the newly inserted boundary. (It is at this point that the framework can introduce extra, evenly spaced boundaries which can be used to reduce power and lower the sporadic task rejection rate.) The smaller, conflict-free plane will then be passed onto the LTF-M algorithm. This solution ensures that at no point can multiple tasks be given access to a single, shared resource.

As mentioned in the last section a task may execute faster than its fluid schedule, resulting in a task requesting a resource early. There are two possible cases that might be encountered. The first case is when T_2 requests the resource while T_1 is holding it. The outcome is that T_2 will be forced to block its execution until after T_1 releases the lock. There is no problem here, because T_2 is not required to be executing yet. The lock naturally solved the problem.The second case is when T_2 executes significantly faster than T_1 such that it requests the lock before T_1 requests the lock. If T_2 is allowed to acquire the lock first, then the system can no longer guarantee the deadline for T_1. In order to satisfy the hard real-time constraints, the system needs a method for selectively granting locks to the tasks. The task cannot employ the "compare and swap" instruction on the processor as the system would not be able to enforce the desired order. And, directing all lock requests through the operating system introduces overhead.

The solution to the problem is using time based locks. This lock is constructed such that only the scheduler / operating system can set the value of a lock to a non-zero value. Moreover, the value of the lock will correspond to the task ID. In order to use the lock, the task will verify that its ID corresponds to the lock's value. If the values match, then it proceeds with the assurance that it has sole possession of the lock. Otherwise, the task will be required to continually check the value of the lock until the value matches the task ID. When the task wishes to release the lock it will set the lock value to 0. Notice that once the task releases the lock it will not be able to reacquire the lock on its own.

In the system described here, the framework shall be responsible for granting the locks. Because the framework has used the requesting edge of a resource as a T-L_{er} plane boundary, it can be shown that only one task will require a given resource within a plane. If multiple tasks require the same resource in a plane, then the tasks are not schedulable because $t_b \geq t_c \; while \; t_a < t_c$. Therefore, if the tasks were originally implemented with no resource conflicts, the scheduler can set the lock value to the one task that requires the resource.

This time based lock only works because the execution time of each critical section has been well defined. The critical code sections will be strictly ordered according to the task utilization values to ensure that all deadlines are met. The tasks can directly check the value of the lock with a read instruction and release the lock with a write

instruction. Therefore, if the fluid task schedule does not show any resource conflicts then the system will function correctly.

5.5 Running the Scheduler

This section shall focus on evaluating the execution requirement for the scheduler and how to invoke the scheduler while meeting the system's requirements.

The execution requirement of the scheduler can be described by the complexity of the framework and the LTF-M algorithms. With bounds on the computational complexity, the worst case execution time of the scheduler can be determined for a given task set. This is to say $c_{scheduler} \leq k * O(\cdot)$ where k is jointly determined by the hardware implementation and the actual code implementing the scheduler. Using the previously derived complexity of the LTF-M algorithm and the complexity of the Framework algorithm the total scheduler complexity is $O(\, n * r \lg(r) + n \lg(n)\,)$ where r is the number of locks in the system. That means $c_{scheduler} \leq k * O(\, n * r \lg(r) + n \lg(n)\,)$ for each plane. This bound on the worst case execution time for the scheduler, allows the system to schedule the scheduler as a task.

The only real-time constraint on the scheduler is that it finishes execution before the start of the T-L_{er} plane. The assumption hitherto has been that the scheduler ran between the end of the previous plane and start of the subsequent plane. However, the power consumption can be reduced by running the scheduler in the background. The system should set $d_{scheduler} = t_{f,previous\ plane}$. By running the scheduler in the previous plane it ensures that it completes the schedule before the next plane begins, while providing the largest possible deadline without overlapping the execution periods of any scheduler invocation. In this case the scheduler appears to be a recurring sporadic task. In each plane the task arrives with unique $c_{scheduler}$ and $d_{scheduler}$ for the plane.

The average power consumption for running the scheduler in the previous plane is lower than running the scheduler between the planes. This can be shown by equations (3) and (4). The average power required for running the scheduler between planes is

$$P_{between,avg} \propto C * f_{\max}^3 * \frac{\left(m * U_{orig}^3 * \frac{t_f^3}{\left(t_f - c_{scheduler}\right)^2} + c_{scheduler} \right)}{t_f}.$$

The power required for the scheduler itself is based on one processor operating at f_{max} for $c_{scheduler}$ time. The execution of both the task set and scheduler must happen within the original t_f for the plane.

However, to calculate the average power when running the scheduler in the previous plane requires the assumption that the work required to schedule the next plane is the same as the work required for the current plane. Setting the work requirements to be equal creates a fair comparison with $P_{between,avg}$ as both cases execute the same amount of work within the same overall timeframe. The power required for running the scheduler in the previous plane is

$$P_{previous,avg} \propto C * f_{max}^3 * m * \left(U_{orig} + \frac{1}{m} * \frac{c_{scheduler}}{t_f} \right)^3.$$

The equation shows that in the ideal case the power consumed in the task set increases in the same fashion as adding a sporadic task to the system.

When $c_{scheduler} = 0$ the average power consumed by each approach is going to be identical. As $c_{scheduler}$ moves towards its upper bound of t_f both equations monotonically increase. The difference in the approaches lies in what the equations increase towards. Taking the limits of both equations

$$\lim_{c_{scheduler} \to t_f} P_{between,avg} = \infty$$

$$\lim_{c_{scheduler} \to t_f} P_{previous,avg} = m * \left(U_{orig} + \frac{1}{m} \right)^3.$$

Since $P_{between,avg} = P_{previous,avg}$ when $c_{scheduler} = 0$ and they both monotonically increase, $P_{between,avg} \geq P_{previous,avg}$. It is more power efficient to run the scheduler inside the previous T-L_{er} plane than to run it between the planes.

6 Unique Solutions for Unique Problems

The scheduler developed up to this point is the high-reliability scheduler. It optimizes the task set execution for the task's worst case execution, assuming that the task always requires c_i execution time during the period. Consequently, it sets the task to have a fluid schedule where $u_i = c_i/d_i$. The system adjusts the power reactively, only adjusting once certain that the task is performing faster.

In an ideal situation each task in the system would execute with $u_i = c_{i,actual}/d_i$ corresponding to the minimum speed required to finish the task by the deadline. The problem with this scenario is that the value $c_{i,actual}$ cannot be known in advance. The solution is to use a predicted value $c_{i,predict}$ instead, which is assumed to be relatively close to $c_{i,actual}$. The trade-off is that an incorrect value may cause the system to consume more power than the high-reliability approach. This section will extend the previously developed high-reliability scheduler to proactively reduce system power.

Ideally the scheduler would know the value $c_{i,actual}$ before the task began execution. But the reality is that this value cannot be known in advance with complete certainty. No prediction is completely accurate. Therefore, the scheduler must be capable of optimizing the task execution based on an inaccurate prediction.

The scheduler cannot trust the inaccurate prediction without inheriting the potential risk of missing a deadline. The optimization must be made while ensuring that it is possible for T_i to execute for c_i by the deadline, $\int_0^{d_i} u_{i,actual} dt = c_i$. The task utilization does not need to be constant during execution, provided that on average $u_{i,actual} = u_i$. The resulting schedule is similar to the stochastic model depicted in Fig. 1. If the task finishes execution early it will require less power than the WCET

model. The scheduler must ensure that it is possible for the worst case execution to be completed exactly at the deadline.

At this point the scheduler is provided with a predicted execution time $c_{i,predict}$. For the scheduler to set a task utilization it must generate a deadline $t_{i,predict}$ for completing the predicted amount of execution. Using these two values the scheduler can execute T_i with $u = c_{i,predict}/d_{i,predict}$ from time 0 to $t_{i,predict}$. If the task does not complete by $t_{i,predict}$ then it will set u higher for the remaining execution.

The scheduler must make a correct decision when it selects $t_{i,predict}$. The value should be large enough to minimize u between time 0 to $t_{i,predict}$ while ensuring that in the latter half of execution $u = (c_i - c_{i,predict})/(d_i - t_{i,predict}) \leq 1$. It must remain possible for a processor to execute the remaining work. This results in an upper bound for the value $t_{i,predict} \leq d_i + c_{i,predict} - c_i$. The scheduler can also develop a lower bound on the predicted deadline. The lower bound provided by the high-reliability design is $t_{i,predict} \geq (d_i * c_{i,predict})/c_i$. As long as the predicted deadline meets these bounds the deadline can be met. Additionally when the $c_{i,predict} \geq c_{i,actual}$ the system power is at least as low as the high-reliability design.

The location at which $t_{i,predict}$ falls inside of its boundaries determines the reliability of the system. At the lower bound the value $c_{i,predict}$ has no impact on the system. In this case the schedule is non-aggressive and is optimized for the worst case. The result is a schedule identical to the high-reliability design. Conversely when $t_{i,predict}$ is at its upper bound, the accuracy of $c_{i,predict}$ has a significant impact on the system. A correct prediction uses the least power possible while guaranteeing the deadline. Meanwhile an incorrect prediction will require more power than the high-reliability schedule.

This approach is not without its penalty. The average power can be worse with an inaccurate predictor. Even more terrible, the instantaneous power consumption can spike. In the case where $c_{i,actual} > c_{i,predict}$, the utilization suddenly increases at $t_{i,predict}$ corresponding to an increase in the required power. However, if all the tasks share a common deadline, every task is poorly predicted, and each task requires the complete worst case execution time c_i, then the total system power will dramatically increase. The utilization of each task directly before the deadline would be 1. The power required to meet this demand is huge. More importantly it means that if $n > m$ then the system cannot guarantee task set feasibility when it aggressively selects $t_{i,predict}$. When the prediction is at the upper bound the scheduler is non-optimal, reduced to $n \leq m$.

This worst case feasibility scenario only happens when both the value $c_{i,predict}$ is inaccurate and the value $t_{i,predict}$ is selected aggressively. Conversely when $c_{i,predict}$is correct or $t_{i,predict}$ is at its lower bound the system is optimal. Using this bit of intuition the system designer can tailor the system for his needs.

This commercial scheduler is not that different from the high-reliability scheduler. The only significant change is that the new design has the task execute with a variable utilization. The actual utilization value used is selected as a trade between potential power savings and system reliability. As the system is adjusted to reduce power more aggressively the guaranteed feasibility bound reduces from Theorem #1 to $n \leq m$.

7 Conclusion and Future Work

A power efficient, hard real-time scheduler for multiprocessor platforms is developed in this paper. The original goal to develop a system that would be useable in an actual system implementation imposes several constraints on the scheduler. It needs to support the spectrum of real-time tasks: periodic, sporadic, and aperiodic. The scheduling mechanism also needs to be scalable while supporting mutually exclusive, shared resources. In the end all of the goals are achieved while maintaining a low complexity of $O(\, n * r \lg(r) + n \lg(n)\,)$ where n is the number of tasks and r is the number of mutually exclusive, shared resources in the system.

The solution is also an optimal scheduler when the lower bound of $t_{i,predict}$ is chosen. In this case it optimizes the schedule for minimum power during the worst case execution scenario. However, the scheduler allows for more aggressive power reductions according to the needs of the system designer. The trade-off is that the feasibility bound is reduced.

This scheduler is useful for an actual implementation. It takes into consideration that processor speeds should be changed sparingly, context switches are minimized, and it takes its own execution time into consideration. The future work is to implement and test this scheduler on an actual hardware platform and get feedback from industry's real-time community.

Acknowledgements This work is partly supported by the National Science Foundation under Grant No. CCF-0541403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8 References

1. Chen, J., Hsu, H., Chuang, K., Yang, C., Pang, A., Kuo, T.: Multiprocessor Energy-Efficient Scheduling with Task Migration Considerations. In: 16th Euromicro Conference on Real-Time Systems. pp. 101--108. IEEE Computer Society, Washington DC (2004)
2. Gruian, F.: Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS Processors. In: 16th International Symposium on Low Power Electronics and Design. pp. 46--51. ACM, New York (2001)
3. Malani, P., Mukre, P., Qiu, Q., W.: Adaptive Scheduling and Voltage Scaling for Multiprocessor Real-Time Applications with Non-Deterministic Workload. In: DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 652--657. ACM, New York (2008)
4. Chen, S., Hsueh, C.: Optimal Dynamic-Priority Real-Time Scheduling Algorithms for Uniform Multiprocessors. In: Proceedings of the 2008 Real-Time Systems Symposium. pp. 147--156. IEEE, Washington DC (2008)
5. Funk, S., Goossens, J., Baruah, S.: On-line Scheduling on Uniform Multiprocessors. In: Proceedings of the 2001 Real-Time Systems Symposium. pp. 183--192. University of North Carolina at Chapel Hill, North Carolina (2001)