Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis

Rony Ghattas and Alexander G. Dean
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
NC State University
Raleigh, NC 27695
{rony_ghattas,alex_dean}@ncsu.edu

Abstract

Using preemption threshold scheduling (PTS) in a multi-threaded real-time embedded system reduces system preemptions and hence reduces run-time overhead while still ensuring real-time constraints are met. However, PTS offers other valuable benefits. In this paper we investigate the use of PTS for hard real-time system with limited RAM. Our primary contribution is to prove the optimality of PTS among all preemption-limiting methods for minimizing a system's total stack memory requirements. We then discuss characteristics of PTS and show how to reduce average worst-case response times. We also introduce a unified framework for using PTS with existing fixed-priority (e.g. rate- or deadline-monotonic), or dynamic-priority scheduling algorithms (e.g. earliest-deadline first).

We evaluate the performance of PTS and our improvements using synthetic workloads and a real-time workload. We show PTS is extremely effective at reducing stack memory requirements. Our enhancements to PTS improve worst-case response-times as well.

1. Introduction

Complex real-time embedded applications often use a *real-time operating system* (RTOS). Most operating systems, however, employ *fully-preemptive* schedulers, which can have excessive preemption overheads. In addition to direct preemption overheads (e.g. CPU time and energy), significant memory support is usually needed. Many low-end RTOS's statically allocate dedicated space in RAM for each of the application's task's stacks (e.g. uC/OS-II, FreeRTOS, AvrX, etc.) which can be prohibitively expensive for systems with little RAM or many tasks [4, 5, 15]. This stack space must be large enough to accommodate worst-case function call nesting, local variable allocation, and possibly the task control block and context. Many kernels also allocate additional space in each task's stack area for servicing interrupts (e.g. uC/OS-II and FreeRTOS)[1].

Many high-volume embedded systems are built around low-end *commercial off-the-shelf* (COTS) microcontrollers because of their low cost. These devices may have very little RAM. Even in systems with larger amounts of RAM, running out of memory can be a common problem. Embedded systems tend to evolve over time, with each new generation of software accreting a new layer onto the existing code base. Hence, over a long enough period of time data memory requirements can easily exceed available memory, making RAM a precious resource. This makes the use of real-time kernels for low-end embedded systems particularly hard if not impossible.

These issues and problems inherent in preemptive real-time kernels have led to the emergence of several methods and standards for designing more efficient kernels with reduced preemption overheads. As the main consumer of low-end microcontrollers, the automotive industry was the first to realize the need for more efficient operating systems by developing the OSEK/VDX standard[2]. Several OSEK/VDX-compliant real-time kernels have been developed. LiveDevices (Realogy) [8,9] introduced several variations of their *real-time architect* (RTA) operating system that utilizes the *single-shot execution* (SSX) model which enables tasks to share a single stack while dividing them into *mutually non-preemptive groups*. Other OSEK/VDX-compliant kernels were also developed to be more efficient in some way [7, 18]. Outside of the automotive industry, other efficient RTOS design techniques have been developed. One very promising technique is *Preemption Threshold Scheduling* (PTS) presented in ThreadX[10], a commercial real-time kernel by Express Logic Inc, and investigated initially by Wang and Saksena [23,25]. PTS tries to minimize preemptions as much as possible while preserving the system's schedulability.

Minimizing preemptions to reduce overhead and im-

[1]Some kernels use more efficient techniques for servicing interrupts like emulating a separate interrupt stack if not supported in hardware like AvrX[4].

[2]The OSEK/VDX standard includes specifications for embedded operating systems, communication subsystems, and embedded network management systems.

prove the system's memory utilization is an ad-hoc process which has not been adequately examined to date. In this paper we analyze and enhance PTS. First, we show that PTS leads to the optimal stack space utilization attainable by any known priority-driven scheduling algorithm (e.g. RM, DM, EDF, etc). Hence, given a particular real-time application, we provide the application engineer with a limit on the amount of memory that can be saved by limiting preemptions while not violating any of the workload's real-time constraints. Second, we build a framework for PTS that naturally applies to both fixed-priority and dynamic-priority schemes. Third, we discuss some characteristics of PTS, including undesired side-effects of preemption limiting in general (e.g. deteriorated system responsiveness and robustness), and show how some can be reduced. Finally, through simulations, we quantify application characteristics (e.g. workload utilization, type of scheduling scheme used, etc.) that affect PTS to help developers of real-time applications.

This paper is divided into six sections. In section 2 we present a brief overview of some of the work addressing preemption threshold scheduling and how it relates to our work. In section 3 we present the background material this paper is based on, and the terminology used throughout this study. Section 4 contains the main theoretical results of this paper: proving the optimality properties of PTS and discussing how to minimize some undesirable side-effects. In section 5 we present a case study and simulations to assess workload characteristics that affect the effectiveness of PTS. Finally, in section 6 we present our conclusions.

2. Related Work

The term *preemption threshold scheduling* (PTS) was first coined by Wang and Saksena[23,25] after they investigated the concept of *preemption thresholds* introduced by Express Logic Inc. in their real-time kernel ThreadX[10]. PTS results in a dual-priority scheme where each task has a nominal priority, which it uses to preempt other tasks, and a preemption threshold which is its effective priority while executing. The real-time analysis of this dual-priority scheme was pioneered by Wang et al.[23, 25] and by Davis et al.[9]. In these studies, preemption between tasks was limited to occur only when necessary to maintain system schedulability. Tasks that run non-preemptively with respect to each other can be mapped into the same run-time thread and share the same run-time stack minimizing the memory requirements and other preemption overheads. Our work builds directly on Wang's et al. work by proving the optimality of their preemption threshold assignment method as well as enhancing its timeliness.

Many resource sharing protocols and real-time synchronization schemes have been adapted to PTS[11,13]. Kim et al.[13] showed how the priority inheritance and priority ceiling protocols could be used with PTS when shared resources are present. Gai et al[11] showed how Baker's stack resource policy [3] can be extended to support PTS, resulting in the *stack resource policy with preemption thresholds* (SRPT). The SRPT inherits all of the properties of the SRP in the sense that it prevents priority inversions, deadlocks, as well as bounds the maximum time any task can be blocked.

Since dynamic-priority task scheduling schemes can be much more efficient than static-priority schemes, it is of particular interest to know if PTS applies to these schemes. Gai's [11] work enables the use of PTS in dynamic-priority schemes through the use of the *preemption levels* concept, also introduced earlier by Baker[3], which enable static analysis of dynamic-priority systems. Another dynamic preemption threshold scheme was presented by He et al.[12] which uses an earliest-deadline-first algorithm to dynamically change the task priorities as well as their preemption thresholds. Our work on PTS for dynamic-priority schemes builds on Gai et al. and Baker's work since, in contrast to He's et al. work, we enable the system developer to analyze and verify a design at design time while avoiding all the additional run-time overheads of dynamically updating preemption thresholds in addition to priorities. To this end, we modify an existing preemption thresholds search algorithm to apply to both fixed- and dynamic-priority schemes. The search can be done offline to obtain the optimal preemption threshold assignment which remains fixed at run-time (incurring no overhead).

Many other studies investigated PTS and presented extensions to its basic concepts. Building on Wang's work for fixed-priority schemes, Regehr[20] presented two abstractions that can be used to force the PTS scheduler to group tasks into non-preemptive groups. Regehr also investigated the presence of WCET uncertainty and presented algorithms for finding fault-tolerant PTS schedules. The incorporation of PTS into many real-time design settings was also addressed. Kim et al.[14] showed how PTS can be used with dynamic voltage scaling to render efficient schedules that optimize the energy usage of the system. Saksena et al.[22], and Wang et al[23] also addressed using PTS in a real-time object oriented framework, showing how real-time object oriented models can be synthesized automatically into a real-time task set implemented in a fixed-priority PTS scheme.

In summary, PTS provides real-time systems developers a method to design efficient real-time schedules in a variety of settings. Our work on PTS is complementary to, or can be used in combination with, most of the other known methods and schemes for using PTS to minimize the overheads of preemptions and obtain more efficient real-time schedules.

3. Terminology and Background

In this section we introduce the system model. We then present previous work in scheduling (both fixed- and dynamic-priority) and resource sharing for PTS, recast into our unified framework.

3.1. Real-Time System Model

A real-time *task*, denoted by T, is an independent thread of execution that competes with other tasks for processor time and other resources. A task is invoked infinitely many times and each invocation results in a single execution which we call a *job*. We will denote the i^{th} task in the workload by T_i and denote the j^{th} job of the i^{th} task by J_{ij}. Every job J_{ij} is characterized by a computation time c_{ij}, and an absolute deadline d_{ij}. Moreover, associated with every job J_{ij} is the amount of stack space s_{ij} required by the job to save its local variables, return addresses for procedure calls, as well as its own context if necessary[3]. We also associate with each job J_{ij} a unique priority $\pi_{ij} \in \{1, 2, \ldots, N\}$ such that contention for resources is resolved in favor of the job with the highest priority that is ready to run.

We let $\mathcal{R} = \{\rho^1, \rho^2, ..., \rho^k\}$ be the set of shared resources. Tasks can access mutually exclusive resources through critical sections. We define ξ^k_{ij} as the critical section of job J_{ij} on the k^{th} resource ρ^k. That is, job J_{ij} will need to access the shared resource ρ^k in a mutually exclusive manner through its critical section ξ^k_{ij}. We denote the maximum time duration of this critical section by ω^k_{ij}.

Tasks can be *periodic* or *sporadic*. If T_i is periodic, the period P_i specifies a constant interval between arrival times of any two consecutive jobs, and if it is sporadic, P_i specifies a minimum interval between job arrivals. We say task T_i has a *Worst-Case Execution Time* (WCET) of C_i time units if all jobs of T_i can take no longer than C_i time units to execute, and say task T_i has a maximum stack space requirement of S_i units if the stack space required by all jobs is no larger than S_i units. Moreover, we say that T_i has a relative deadline of D_i time units if all jobs must complete execution no more than D_i time units after arrival.

Hereafter, we use the notion $\mathcal{T} = \{T_i = \langle P_i, D_i, C_i, S_i \rangle | i = 1, 2, \ldots, N\}$ to denote the set of N tasks composing our real-time workload. We will use the tuple $(\mathcal{T}, \Pi)$ to denote the workload $\mathcal{T}$ along with some priority assignment Π.

3.2. Scheduling Policies

A scheduling policy dictates the order in which different jobs use the processor and how different requests should be serviced. *Priority-driven* scheduling policies are a large subset of all scheduling policies and are used by most real-time kernels. A priority-driven scheduling policy is a mapping $\Pi : \mathcal{T} \to \{1, 2, 3, \ldots, N\}$ governing the execution of jobs such that contention for resources is always resolved in favor of the job with the highest priority that is ready to run. An essential property of a scheduling policy is *preemptability*. We say a scheduling policy is *fully-preemptive* if a currently executing task can be preempted by a ready-to-run higher priority task. On the other hand, we say a scheduling policy is *fully-non-preemptive* if the ready-to-run task has to wait for the currently executing lower priority task to release the processor voluntarily.

Between the two boundary cases of fully-preemptive and fully-non-preemptive scheduling policies lies preemption threshold scheduling or PTS. With PTS, a real-time system with some priority assignment $(\mathcal{T}, \Pi)$ is assigned an additional mapping $\Gamma : (\mathcal{T}, \Pi) \to \{1, 2, \ldots, N\}$ (i.e. each task is assigned a *preemption threshold*, denoted by γ_i in addition to its priority π_{ij} with the essential property that $\gamma_i \geq \pi_{ij} \forall j$). When the task begins execution, its priority is raised to its preemption threshold. In this way, all the tasks with priorities less than or equal to the preemption threshold of the executing task cannot preempt it. PTS effectively creates groups of *mutually non-preemptive* tasks that are not allowed to preempt each other.

It is easily seen that fully-preemptive and fully-non-preemptive scheduling policies are special boundary cases of PTS. By assigning the preemption threshold of each task equal to its priority, PTS simplifies to a fully-preemptive scheduling policy. By assigning the preemption threshold of each task equal to the system's highest priority, PTS simplifies to a fully-non-preemptive policy.

Another scheduling policy characteristic is that it can be static or dynamic. A *fixed-priority* scheduling policy is mapping Π that does not change at run time; otherwise it is a *dynamic-priority* scheduling policy. A basic property of all fixed-priority schemes is that all jobs belonging to the same task are assigned the same priority (i.e. $\Pi(J_{ij}) \equiv \Pi(T_i) = \pi_i \ \forall j$). Hence, in a fixed-priority scheme we can speak of a "task's" priority instead of a "job's" priority. Classical examples of fixed-priority scheduling algorithms are Rate Monotonic (RM), Deadline Monotonic (DM), and Audsley's optimal priority assignment algorithms [2, 17].

Unlike fixed-priority scheduling, in a dynamic-priority scheme different jobs belonging to the same task can be assigned different priorities dynamically. The most popular dynamic priority scheduling policy is the *earliest deadline first* (EDF) algorithm [17]. In EDF scheduling, the priority of each job is assigned dynamically to be inversely proportional to job's absolute deadline. Below we discuss some of the basic schedulability tests for both of these schemes.

3.3. Fixed-Priority Schedulability Analysis

Given a real-time workload and some priority assignment, $(\mathcal{T}, \Pi)$, we would like to know if the workload is schedulable (i.e. all tasks in the workload will meet their deadlines). For fixed-priority scheduling policies, schedulability can be analyzed using *level-i busy period analysis*[16, 24]. To this end, a bound is computed on the response times of all jobs of a task by essentially simulating some worst-case scenario that jobs can experience. Such a bound is referred to as the *worst-case*

[3]The amount of stack space needed by each job can be computed and bounded with existing static analysis tools[21].

response time (WCRT) for the task and denoted by W_i. If the WCRT each task is no larger than its respective deadline, the workload is said to be *schedulable*.

Consider a real-time workload with some predefined fixed-priority assignment $(\mathcal{T}, \Pi)$, and let $T_i \in \mathcal{T}$ be any task. Let $\mathcal{HP}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with priorities larger than that of T_i (i.e. $\mathcal{HP}(T_i) = \{T_j \in \mathcal{T} | \pi_j > \pi_i\} \subset \mathcal{T}$). Similarly, let $\mathcal{LP}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with priorities smaller than that of T_i. The WCRT of a task will occur if one of its jobs is released at the same time with a job from every higher priority task. Hence, the WCRT of a task can be obtained by considering the response-time of a single job that is released under this worst-case scenario. This WCRT can be computed using the following recursive equations [24]:

$$w_i(q) = q \cdot C_i + \sum_{T_j \in \mathcal{HP}(T_i)} \left\lceil \frac{w_i(q)}{P_j} \right\rceil C_j \tag{3.1a}$$

where $w_i(q)$ denotes the length of a busy period for task T_i with q jobs (instances) of T_i included in the busy period. The worst-case response time W_i for task T_i is then computed using the following expression:

$$W_i = \min_{q \in \{0,1,2,\ldots\}} w_i(q) \tag{3.1b}$$

This iteration over increasing values of q stops if $w_i(q) \leq q \cdot P_i$. A task set is schedulable fully-preemptively if and only if $W_i \leq D_i$ for all $i \in [1, N]$. Hence, these conditions are a necessary and sufficient condition for schedulability in a fixed-priority scheme.

However, a task might experience a different kind of a worst-case scenario when PTS is used. That is, in addition to interference from higher priority tasks in $\mathcal{HP}(T_i)$, with PTS, a task T_i may also be blocked by a lower priority task $T_j \in \mathcal{LP}(T_i)$ if $\pi_i \leq \gamma_j$. In this case, the WCRT of a task happens when the task with the longest WCET that has lower priority but higher preemption threshold is released one clock cycle before T_i. In this case, the worst-case start time $\mathcal{S}_i$, the worst-case finish time $\mathcal{F}_i$, and the worst-case response time W_i of T_i are given by the following three equations, respectively[20, 25]:

$$\begin{aligned} \mathcal{S}_i(q) &= B_i + q \cdot C_i \\ &+ \sum_{T_j \in \mathcal{HP}(T_i)} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{P_j} \right\rfloor\right) C_j \end{aligned} \tag{3.2a}$$

$$\begin{aligned} \mathcal{F}_i(q) &= \mathcal{S}_i(q) + C_i \\ &+ \sum_{\substack{T_j \in \mathcal{T} \\ \pi_i > \gamma_j}} \left(\left\lceil \frac{\mathcal{F}_i(q)}{P_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{P_j} \right\rfloor\right)\right) C_j \end{aligned} \tag{3.2b}$$

$$W_i = \max_{q \in \{0,1,\ldots,\lfloor L_i/P_i \rfloor\}} (\mathcal{F}_i(q) - q \cdot P_i) \tag{3.2c}$$

where L_i is the longest level-i busy period and is given by the following:

$$L_i = B_i + \sum_{T_j \in \mathcal{HP}(T_i)} \left\lceil \frac{L_i}{P_j} \right\rceil C_j \tag{3.2d}$$

while B_i denotes the worst-case blocking T_i can experience due to tasks in $\mathcal{LP}(T_i)$ with preemption thresholds equal to or larger than the task's priority and is given by the following:

$$B_i = \max_{\substack{T_j \in \mathcal{LP}(T_i) \\ \pi_i \leq \gamma_j}} (C_j - 1) \tag{3.2e}$$

3.4. Dynamic-Priority Schedulability Analysis

In a dynamic-priority scheme, a condition for the schedulability of a real-time system is given by the following *EDF schedulability test*[17]:

$$\sum_{i=1}^{N} \frac{C_i}{\min(D_i, P_i)} \leq 1 \tag{3.3}$$

If the above condition is satisfied, we conclude that our workload is schedulable with the EDF scheme. We note, however, that if it is not satisfied, then the conclusion we may draw depends on the relative deadline of the tasks. If $D_i \geq P_i$ for all i, then equation (3.3) reduces to the well known *EDF utilization test* which is both a necessary and sufficient condition. On the other hand, if $D_i < P_i$ for some i, equation (3.3) is only sufficient and we can only conclude that the system may not be schedulable.

To use PTS with a dynamic-priority scheme, however, the system preemption relations need to be known at design time (i.e. which tasks/jobs can preempt which tasks/jobs). To this end, Gai et al. [11] showed that the concept of *preemption levels* presented earlier by Baker[3] can be used. Preemption levels present a static mapping that enables offline analysis of dynamic-priority scheduling schemes. A preemption level, denoted by λ_i, for task T_i is assigned statically at design time similar to the task's priority in the case of fixed-priority schemes (i.e. the preemption levels of all jobs belonging to a task are the same). As explained by Baker, the preemption level mapping is arbitrary as long as it is assigned to satisfy one single property: *A job of T_i is not allowed to preempt a job of T_j unless it has higher priority and $\lambda_i > \lambda_j$.* Under the EDF scheduling policy, the previous property has been verified if preemption levels are assigned inversely proportional to the tasks' periods (i.e. $\Lambda(T_i) = \lambda_i \propto \frac{1}{P_i}$)[3]. After preemption levels are assigned, the system can be analyzed statically as in the fixed-priority case and preemption thresholds can be assigned. Hence, through the use of preemption levels, one can transform a dynamic-priority system $(\mathcal{T}, \Pi)$ into $(\mathcal{T}, \Lambda)$ with the mapping Λ being static, enabling design time analysis similar to fixed-priority schemes.

Given a dynamic-priority scheduled workload $(\mathcal{T}, \Pi)$, we can use the preemption level mapping Λ to transform it into the statically analyzable system $(\mathcal{T}, \Lambda)$. In this context, let $\mathcal{HL}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with *preemption levels* that are larger than that of T_i (i.e. $\mathcal{HL}(T_i) = \{T_j \in \mathcal{T} | \lambda_j > \lambda_i\} \subset \mathcal{T}$). Similarly, let $\mathcal{LL}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with preemption levels smaller than that of T_i. Based on Baker's work, we develop the following expression for the *maximum blocking* that any task T_i can experience while maintaining its schedulability [3, 11]:

$$B_i^{max} = \left(1 - \sum_{T_j \in \mathcal{HL}(T_i)} \frac{C_j}{\min(D_j, P_j)} \right) P_i \qquad (3.4a)$$

As with fixed-priority schemes, we can now require that the blocking B_i experienced by $T_i \in (\mathcal{T}, \Lambda)$ due to tasks with lower preemption levels (which in some sense is like having a lower priority) be no larger than the maximal blocking that the task can handle without losing schedulability (i.e. $B_i \leq B_i^{max} \; \forall i$). The blocking term B_i in this case is given as:

$$B_i = \max_{\substack{T_j \in \mathcal{LL}(T_i) \\ \lambda_i \leq \gamma_j}} (C_j - 1) \qquad (3.4b)$$

3.5. Resource Sharing Protocols

In the previous sections, tasks were assumed to be independent. However, there are resources (e.g. global variables, buffers, hardware devices, etc.) that tasks must access in a mutually exclusive manner. In fully-preemptive and semi-preemptive schemes like PTS, the mechanisms that enforce mutual exclusion can block a task until another task releases the resource. [4]

Many resource sharing protocols have been proposed: the *priority inheritance protocol* (PIP), the *priority ceiling protocol* (PCP), the *stack resource policy* (SRP), etc[17]. The PIP prescribes that if a higher priority job becomes blocked by a lower priority one due to a shared resource, the job that is causing the blocking should execute with a priority which is the maximum of its own nominal priority and the priority of the jobs that it is currently causing the blocking (i.e. it should inherit the priority of the higher priority task). However, the PIP does not prevent deadlocks. In addition, a job can be blocked multiple times.

The PCP solves the PIP problem by adding to each shared resource ρ^k a priority, called the *priority ceiling* of the resource and denoted by $ceil(\rho^k)$. The priority ceiling is an upper bound on the priority of any job that may lock the resource. A job that would require a particular resource is then prevented from entering its critical section unless its priority is higher than the priority ceiling of all the shared resources it might acquire. This prevents the deadlocks and guarantees that a job can be blocked at most once.

Similar to the PCP, the *Stack Resource Policy* (SRP) was developed by Baker[3] to enable real-time tasks to share mutually exclusive resources while preventing deadlocks. The SRP can be used with dynamic-priority assignment scheduling as well as static-priority scheduling. The SRP ensures that once a task starts execution, it cannot be blocked until completion. A task can only be preempted by higher priority tasks. However, the execution of a task T_i with the highest priority in the system could be delayed by a lower priority task, which is locking some resource, and has raised the system ceiling to a value greater than or equal to the priority level λ_i. Similar to the PCP, this delay is called the *resource blocking time* and denoted by B_i^r.

The blocking time due to shared resources B_i^r can now be used to modify the schedulability conditions for fixed-priority and dynamic-priority schemes. To this end, it can be shown that the maximum blocking time a job can experience due to a shared resource can be calculated as the longest critical section ξ_{lj}^k of tasks with lower preemption levels (which are equivalent to priorities in the case of fixed-priority scheduling policies), but with a ceiling greater than or equal to the preemption level of T_i:

$$B_i^r = \max_{T_l \in \mathcal{T}, \forall j} [\omega_{lj}^k | \lambda_i > \lambda_l \wedge \lambda_i \leq ceil(\rho^k)] \qquad (3.5)$$

Equation (3.5) can then be used to modify the schedulability conditions for fixed-priority and dynamic-priority schemes. To this end, the blocking terms in equations (3.2e) and (3.4b) can be taken as the maximum of the blocking due to lower preemption level tasks with higher preemption thresholds and the blocking due to shared resources. As explained by Gai et al.[11], this enables the use of PTS in the presence of shared resources for preventing priority inversions, deadlocks, and enabling tasks synchronization. From an implementation point of view, the SRPT also allows tasks to share a unique stack since a task is never blocked due to a shared resource, it simply cannot start executing if its preemption level is not high enough.

Blocking due to self suspension is not an issue either since tasks belonging to different non-preemptive groups can have different stack spaces. Hence, if a task delays itself and later resumes execution, it will not corrupt its stack space because it only shares the same stack space with other tasks that cannot preempt it. Tasks that are allowed to preempt the suspended task, on the other hand, are located in different areas of memory and will not cause any stack corruption.

[4]Note that mutual exclusion is not a problem for fully-non-preemptive scheduling schemes since tasks cannot be interrupted while accessing a resource.

4. PTS: Memory Optimality and Response Time Analysis

In this section we prove PTS yields optimal system stack memory use when used with either a fixed- or dynamic-priority based preemption scheme. We then present an enhancement of PTS which reduces task worst-case response times.

4.1. Stack Memory Optimality

In this section we assume that we are given the tuple $(\mathcal{T}, \Lambda)$ where the preemption level mapping has been already assigned as explained earlier for both the fixed- and the dynamic-priority schemes. We would like to find a preemption threshold mapping Γ that is feasible and in some sense optimal. A preemption threshold assignment Γ is *feasible* if and only if the system is schedulable according to the conditions presented earlier for fixed-priority and dynamic-priority schemes. Hereafter, the set of all feasible preemption threshold assignments for the system $(\mathcal{T}, \Lambda)$ will be denoted by $\mathcal{G}(\mathcal{T}, \Lambda)$. A particular feasible assignment of special interest is defined below:

Definition 4.1. *(Identity Preemption Threshold Assignment) We define the identity preemption threshold assignment, $\Gamma^I \in \mathcal{G}(\mathcal{T}, \Lambda)$, as the preemption threshold assignment where all tasks have been assigned preemption thresholds that are equal to the tasks' preemption levels (i.e. $\gamma_i = \lambda_i$ for all $i \in [1, N]$).*

Since enhancing schedulability of a system with PTS has already been addressed [23, 25], in this paper we only consider systems that are schedulable in a fully-preemptive manner with some predefined preemption levels. These preemption levels (derived from the tasks' priorities and parameters as explained earlier) are pre-assigned according to some scheduling algorithm (e.g. RM, DM, EDF, etc), so the identity assignment is always known. Moreover, $\mathcal{G}(\mathcal{T}, \Lambda)$ is never empty since it will at least contain the identity assignment.

Given the identity assignment Γ^I, other assignments that optimize the system in some sense need to be found. Wang et al. [25] developed a heuristic that can always find a feasible preemption threshold assignment if it exists with a time complexity[5] of $O(N^2 \cdot q(N))$. We will call this algorithm the *Maximal Preemption Threshold Assignment Algorithm* (MPTAA) because it will always find the preemption threshold assignment that is *larger* than any other feasible preemption threshold assignment, as shown by Chen et al[6]. In this context, given two preemption threshold assignments, $\Gamma, \Gamma' \in \mathcal{G}(\mathcal{T}, \Lambda)$, we say that Γ is larger than Γ', and denote it by $\Gamma \succeq \Gamma'$, if and only if all member preemption thresholds of Γ are equal to or greater than the corresponding preemption thresholds of Γ' (i.e. $\gamma_i \geq \gamma'_i \; \forall i$). The largest of all preemption threshold assignments is of particular interest and is defined as follows:

[5]The function $q(N)$ is dependent on the priority assignment policy used to check the schedulability of a task on line 7 of the MPTAA[3, 6].

Algorithm 1 $MPTAA(\mathcal{T}, \Pi)$

```
1:  $\Gamma = \Gamma^I$    /* initialize preemption threshold to identity assignment */
2:  for $i = N$ down to 1 do
3:      $j = i + 1$;
4:      while ( $schedulable == TRUE$ and $\gamma_i < N$) do
5:          $\gamma_i = \gamma_i + 1$;
6:          /* check the schedulability of the affected task */
7:          $schedulable = is\_task\_schedulable(T_j)$;
8:          if ($schedulable == FALSE$) then
9:              $\gamma_i = \gamma_i - 1$;
10:         end if
11:         $j = j + 1$;
12:     end while
13:     $schedulable = TRUE$;
14: end for
15: return $\Gamma$;
```

Definition 4.2. *(Maximal Preemption Threshold Assignment) We define the maximal preemption threshold assignment, denoted by $\Gamma^{max} = (\gamma_1^{max}, \gamma_2^{max}, \ldots, \gamma_N^{max}) \in \mathcal{G}(\mathcal{T}, \Lambda)$, as the largest preemption threshold assignment in $\mathcal{G}(\mathcal{T}, \Lambda)$. That is, $\Gamma^{max} \succeq \Gamma$ for all $\Gamma \in \mathcal{G}(\mathcal{T}, \Lambda)$.*

The MPTAA as presented in [25] is shown in algorithm 1. However, line 7 of this algorithm was modified to apply to both fixed-priority as well as dynamic-priority schemes. To this end, the procedure $is\_task\_schedulable()$ will evaluate the schedulability of the system using either level-i busy period analysis for fixed-priority systems (Eqn. 3.2c), or by computing the maximal blocking a task can endure without violating its schedulability for dynamic-priority schemes (Eqn. 3.4a).

The MPTAA was analyzed by Chen et al.[6], and was shown to always find the maximal preemption threshold assignment if one exists[6] (theorem 3 in [6]). Since in our case we always start with a feasible assignment, namely the identity assignment, the maximal preemption threshold assignment always exists (in the worst-case being equal to the identity assignment) and will always be found by the MPTAA.

We now present the main theorem for this section. In proving this theorem, we will denote the total stack space requirements of a system $(\mathcal{T}, \Lambda)$ and a particular preemption threshold assignment Γ by $\mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma)$. We emphasize that the nature of the scheduling scheme (i.e. whether static or dynamic) is irrelevant to our proof and hence will not be mentioned explicitly.

[6]This was only addressed for fixed-priority schemes, but holds equally for dynamic-priority schemes as well.

Theorem 4.1. *Given two real-time systems $(\mathcal{T}, \Lambda, \Gamma)$ and $(\mathcal{T}, \Lambda, \Gamma')$ with preemption thresholds assignments Γ and Γ' in $\mathcal{G}(\mathcal{T}, \Lambda)$ such that $\Gamma \succeq \Gamma'$. The total stack size $\mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma')$ can be no smaller than $\mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma)$.*

Proof. Without loss of generality, let us assume that $\gamma_k = \gamma'_k$ for all $k = 1, 2, \ldots, i-1, i+1, \ldots, N$ and let $\gamma'_i = \gamma_i - 1 < \gamma_i$ for some arbitrary $i \in \{1, 2, \ldots, N\}$. It should be clear that $\Gamma \succeq \Gamma'$ according to our definition. Now let $T_j \in \mathcal{T}$ be the task belonging to the system with a priority (preemption level) π_j (λ_j) chosen such that $\pi_j = \gamma'_i + 1 = \gamma_i(\lambda_j = \gamma'_i + 1 = \gamma_i)$. The existence of T_j is guaranteed by the way we assign preemption thresholds. Now since $\pi_j = \gamma_i(\lambda_j = \gamma_i)$, then T_j is not allowed to preempt T_i with the larger preemption threshold assignment Γ according to the rules of preemption threshold scheduling. On the other hand, T_j is allowed to preempt T_i with the smaller preemption threshold assignment Γ' since $\gamma'_i < \gamma_i = \pi_j(\gamma'_i < \gamma_i = \lambda_j)$. Hence, we have to allocate an additional S_j stack units to accommodate the potential preemption between T_j and T_i if we use the smaller preemption threshold assignment Γ', and therefore:

$$\mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma') = \begin{cases} \mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma) + S_j & \text{if } T_j \text{ preempts } T_i \\ \mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma) & \text{otherwise} \end{cases}$$

Hence, $\mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma) \leq \mathcal{S}_{total}(\mathcal{T}, \Lambda, \Gamma')$ which completes the proof. □

The above theorem and the definition of the maximal preemption threshold assignment leads us directly to the following important corollary.

Corollary 4.2. *The MPTAA finds the preemption threshold assignment with the smallest possible total stack space requirements.*

Proof. Theorem 3 in [6] shows that the MPTAA finds the maximal preemption threshold assignment Γ^{max} with the essential property that $\Gamma^{max} \succeq \Gamma$ for all $\Gamma \in \mathcal{G}(\mathcal{T}, \Lambda)$. Combining this with theorem 4.1 proves this corollary. □

Hence PTS can be used with any scheduling algorithm (RM, EDF, etc.) and always renders the smallest stack space requirements while maintaining the schedulability of the system. Note that the above corrollary also implies that the MPTAA will result in the fewest number of mutually non-preemptive groups. These groups can be mapped to the same run-time thread as was explained by Wang et al. [23]. Nevertheless, minimizing the number of non-preemptive groups alone does not result in minimizing the stack space requirements unless all tasks have the same stack requirement.

4.2. Improving System Responsiveness

One goal of using a preemptive policy is to improve the system's responsiveness to internal or external stimuli. Hence, a real-system developer would prefer to use a particular preemption threshold assignment if it results in the minimal stack size but also improves system responsiveness. The formulation presented in this section enables system developers to make an informed tradeoff between stack space utilization and the degree of responsiveness desired.

Before we continue we need to define a way to measure system responsiveness. A well-suited metric for measuring system responsiveness in a fixed-priority scheme is the *average worst case response time* (AWCRT), computed by averaging the WCRT over all tasks in a system, which we will denote by $AW(\mathcal{T}, \Pi^{stat}, \Gamma)$[7]. We leave as future work the analysis of other potentially useful metrics, such as average WCRT relative to period, or with weightings.

We show in this section that there might exist a preemption threshold assignment different from Γ^{max} that can result in the *same* optimal total stack space requirements while improving the system's responsiveness as measured by the AWCRT. That is, we can find a feasible preemption threshold assignment, that we shall denote by Γ^{opt}, such that the following two properties hold.

$$\mathcal{S}_{total}(\mathcal{T}, \Pi^{stat}, \Gamma^{opt}) = \mathcal{S}_{total}(\mathcal{T}, \Pi^{stat}, \Gamma^{max}) \quad (4.6a)$$

$$AW(\mathcal{T}, \Pi^{stat}, \Gamma^{opt}) \leq AW(\mathcal{T}, \Pi^{stat}, \Gamma^{max}) \quad (4.6b)$$

As an example, a randomly generated system of 5 tasks was used to explore the search path of the MPTAA until it reaches the maximal assignment Γ^{max}. The path is shown in figure 4.1 with all the assignments visited starting from the identity assignment. The MPTAA finds the maximal threshold assignment that minimizes the system stack requirements from 160 to 90 units while maintaining schedulability. However, note the grouping of data points; there are multiple vertically-aligned clusters of points with the same worst-case total stack size but differing AWCRT. The MPTAA reaches each cluster's lowest point (i.e. best AWCRT) first, but then proceeds upward, raising AWCRT without improving the total stack size. Although the MPTAA minimizes the system's stack usage at 90 units, there are *two* other preemption threshold assignments that result in the *same* memory-optimal total stack requirement but with *better* system responsiveness. The best is Γ^{opt}, which has an AWCRT of 41.4 time units, versus 45.2 for Γ^{max}.

The list of preemption threshold assignments given in figure 4.1 can also be used by system developers

[7]For simplicity, we focus in this section on fixed-priority systems. However, it should be emphasized that this treatment is equally applicable to dynamic-priority systems by defining some other metric to measure the system responsiveness.

PT Assignment	Stack	AWCRT
$\Gamma^I = (1,2,3,4,5)$	160	26.6
$\Gamma = (1,2,3,5,5)$	150	28.4
$\Gamma = (1,2,4,5,5)$	150	29.6
$\Gamma = (1,2,5,5,5)$	140	29.6
$\Gamma = (1,3,5,5,5)$	110	31.2
$\Gamma = (1,4,5,5,5)$	110	31.6
$\Gamma = (1,5,5,5,5)$	110	31.6
$\Gamma = (2,5,5,5,5)$	110	38.2
$\Gamma^{opt} = (3,5,5,5,5)$	90	41.4
$\Gamma = (4,5,5,5,5)$	90	43.4
$\Gamma^{max} = (5,5,5,5,5)$	90	45.2

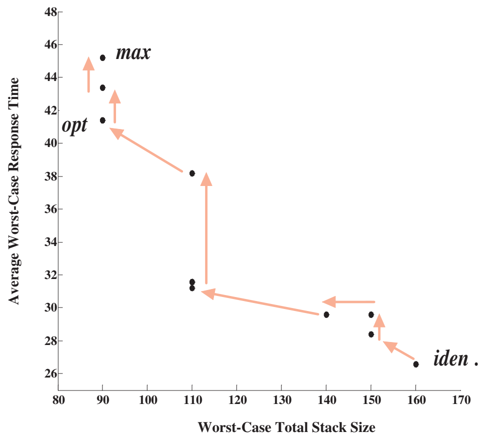

Figure 4.1. Maximal preemption threshold assignment passes case which is optimal for both system response time and total stack size.

who prefer to have higher system responsiveness at the expense of additional stack space. The designer chooses from the list the desired level of responsiveness and uses the corresponding preemption threshold assignment.

To compute Γ^{opt}, we traverse the problem graph backward starting with Γ^{max} and exit as soon as the total stack size starts increasing as shown in Figure 4.1. That is, after each iteration of algorithm 1, the preemption threshold assignment Γ is saved in a list. After Γ^{max} has been found, this list is traversed backwards to determine if the same (optimal) stack size can be obtained with better system responsiveness.

5. Case Studies and Simulations

Section 4 showed that PTS will always render the smallest stack size that will maintain the schedulability of the workload. We now evaluate the impact of PTS on stack space and response time in two ways. First, we use PTS to schedule a real workload developed for controlling an Unmaned Aviation Vehicle (UAV). Second, we use randomly-generated workloads to examine broad trends across a range of design points.

5.1. Paparazzi Benchmark

The "Paparazzi" project of Brisset and Drouin [1] targets a cheap fixed-wing autonomous UAV executing a predefined mission. Nemer et al.[19] used the

Table 1. The real-time tasks composing the Fly-By-Wire benchmark

Name	Frequency	WCEC	Stack
receive_radio	40Hz	14,820	34B
check_failsafe	20Hz	12,477	6B
check_autopilot_values	20Hz	5,680	26B
send_data_to_autopilot	40Hz	5,640	26B
servo_transmit	20Hz	2394	10B
servo_interrupt	_	80	2B
spi_interrupt	_	193	2B
radio_interrupt	_	76	2B

Paparazzi project to develop a real-time benchmark called "PapaBench". Papabench is composed of two workloads with tasks for controlling the servo system, and handling navigation and stabilization. In this section we used PTS to schedule and optimize the servo controller tasks from PapaBench listed in Table 1 with their *worst-case execution cycles* (WCEC). The maximum stack space required by each task was also computed using the *stacktool* by Regher [21]. The workload was designed to be able to run without preemption; many precedence relations exist among the tasks, and the total processor utilization is quite low (less than 10%). For this evaluation, however, we assume that the task precedence relationships and utilization are unknown and hence assume that a fully-preemptive scheduling approach is required. In this case the task set requires 120 bytes for global data and 108 bytes for stack data (detailed in the table). Using PTS to limit preemptions reduces this total stack space requirement from 108 bytes to 34 bytes while maintaining schedulability. This holds for the utilization levels examined (37% to 97%). The total RAM required is reduced by 37%, a significant amount. Indeed, PTS with the MP-TAA provides us with a simple systematic method that can be applied directly to any system independent of the priority assignment policy used. This method provides the real-time system designer with a tool to investigate the minimal memory requirements that maintain the schedulability of system, without which PTS would be error-prone.

5.2. Generic Real-Time Workloads

We next investigate workload characteristics that affect the stack size optimality level achievable through PTS. For example, the optimal stack utilization for some workloads through the use of PTS can be as small as 20% of the stack space utilization required by the fully-preemptive version of the system while others need 80%. We now simulate and analyze randomly generated systems of tasks to better understand PTS.

To cover a wide range of design points, 40,000 systems with 10 tasks each were randomly generated. These were created so 1000 have a utilization of 50%, 1000 have 51% utilization, and so on up to 90%. Task periods have a normal distribution with a mean, $\bar{P}$, of 100 time units and a standard deviation, σ_P, of 25%, 50%, and 75%, respectively. Moreover, task deadlines

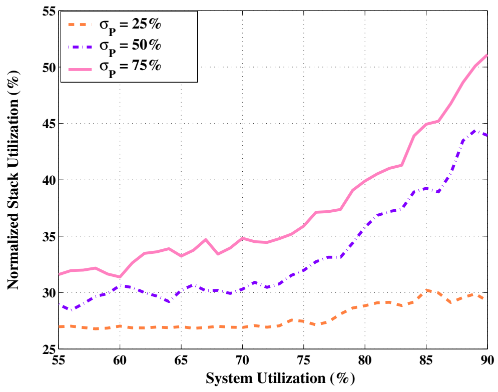

Figure 5.2. Stack space required for PTS with a fixed-priority policy rises with both task period variability and system utilization.

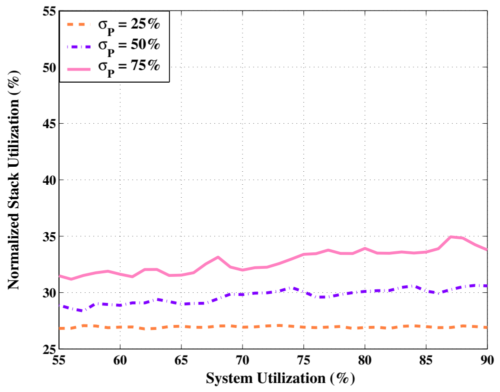

Figure 5.3. Stack space required for PTS with a dynamic-priority policy depends mostly on task period variability.

were set equal to their respective periods (for simplicity, though not necessary). Tasks WCETs were set to incur the required overall system utilization. Task maximum stack space utilizations were chosen from a uniform distribution between 20 and 120 units. All 40,000 systems generated were schedulable with a fully-preemptive policy.

We first investigate the effect of the system utilization on the optimal stack utilization required with PTS. Using the MPTAA, the optimal stack space required by each system was computed and normalized to the stack space required by the fully-preemptive version of the system. The average normalized stack utilizations were then plotted as a function of the overall system utilization and the standard deviation in the task periods. The results are shown in figures 5.2 and 5.3 for the fixed-priority and the dynamic-priority schemes, respectively.

First, consider the fixed-priority scheme in figure 5.2. At low utilizations the system's stack space requirements might be less than 30% of those of a fully-preemptive system. However, at higher utilization levels, the variation in the tasks' periods increasingly affects the savings attainable. With $\sigma_P = 25\%$ the savings are only slightly dependent on the utilization level. On the other hand, as the standard deviation of the periods increases, the savings attainable decrease significantly at higher utilizations. This can be attributed to the fact that for systems with large variations in their periods it is much harder to maintain the system's schedulability while minimizing preemptions. For example, if a system contains two tasks that have a large difference in frequencies, it is difficult, if not impossible, to limit the higher frequency task from preempting the slower while maintaining system schedulability. This becomes very apparent at high system utilizations where there is much less slack time.

Second, consider the dynamic-priority scheme in figure 5.3. The normalized stack space utilizations in this case are much more uniform across all utilization levels. This is because the dynamic (EDF) scheme is much more *adaptive* to the workload characteristics and has a higher *schedulable utilization*[8] than a fixed-priority scheme such as RM. This more-efficient scheduling allows more preemption limiting to occur before schedulability is lost.

Another interesting property is the distribution of the 40,000 systems among the different normalized stack space utilizations levels. To this end, a histogram was constructed showing the percentage of systems versus the normalized (optimal) stack space utilization achievable by each system. Figure 5.4 and figure 5.5 show this distribution for the overall system utilization levels of 60%, 70%, 80%, and 90%, respectively. Again, as can be seen, the workloads scheduled with the dynamic-priority schemes depend less on the system utilization level than those in a fixed-priority scheme.

5.3. System Responsiveness with PTS

Limiting system preemptions has some undesirable side effects, including increasing a task's WCRT. The increase is not constant and depends on workload characteristics. In section 4.2, we discussed reducing the effect of preemption limiting on the WCRT through backtracking.

To investigate this important issue, the workloads generated in the previous section were arranged in order of relative improvement in stack memory requirements. The AWCRT for the workloads was then computed and normalized to the optimal AWCRT of the fully-preemptive version of the system. For example, if the AWCRT of a system with PTS doubles as compared to its fully-preemptive AWCRT, then its normalized AWCRT is 200%. This data was then plotted as shown in figure 5.6 for the MPTAA with and without backtracking.

[8]The *schedulable utilization* of a scheduling algorithm is defined by Liu[17] as the utilization level that guarantees that any system with this utilization can feasibly be scheduled with this algorithm. It is known that EDF has a schedulable utilization of unity as compared to RM with schedulable utilization of $N(2^{1/N} - 1)$ for N tasks.

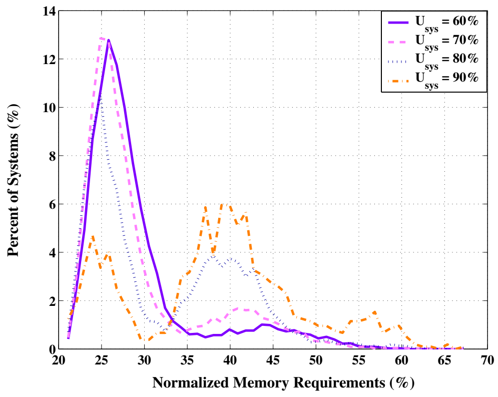

Figure 5.4. PTS and a fixed-priority policy dramatically reduce the stack space required, even for high-utilization systems.

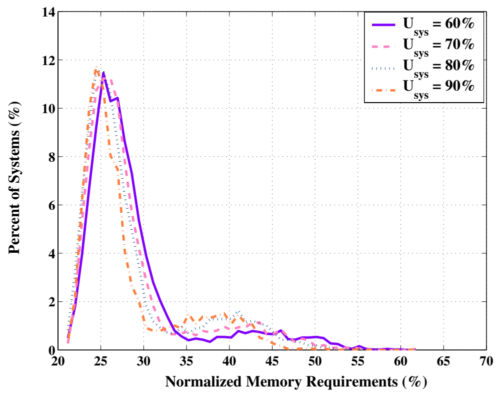

Figure 5.5. PTS and a dynamic-priority policy reduce the stack space required even better than a fixed-priority policy.

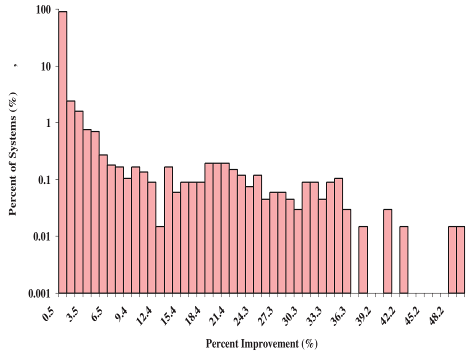

Figure 5.7. The distribution of system-level response time (AWCRT) improvement shows task sets benefit from backtracking, with some showing a dramatic benefit.

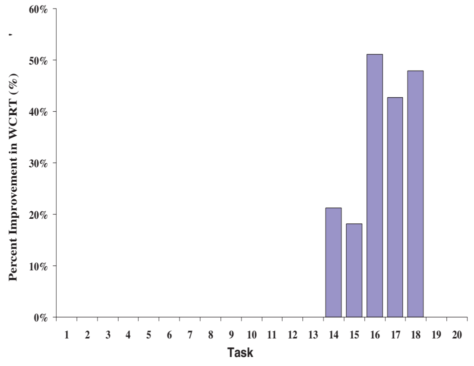

Figure 5.8. A small AWCRT improvement (0.98% for this task set) masks a large task-level WCRT improvement. Five tasks see WCRT reductions from 15% to 50%.

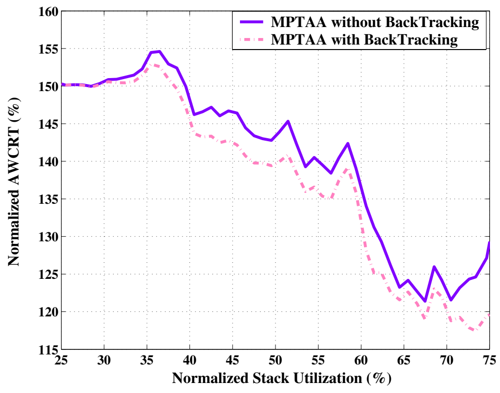

Figure 5.6. System response time (AWCRT) using PTS is improved by back-tracking from the maximal to the optimal preemption threshold assignment.

Figure 5.6 shows two relations. First, the normalized AWCRT is inversely related to the optimal normalized stack utilization; as the system's stack space requirements increase, its normalized AWCRT decreases. This is expected, since systems utilizing more stack space allow more preemptions, resulting in better system responsiveness. Second, backtracking results in greater improvement at higher stack utilizations. When many preemptions are needed for the system, the MPTAA over-constrains some of the tasks unnecessarily even though the optimal stack utilization has been reached and further limiting of preemptability will not succeed in minimizing the stack further.

Finally, we examine the improvement from backtracking the MPTAA. The histogram of Figure 5.7[9] shows that although most of the systems had between 0.5% to 1% improvement only, some systems had a dramatic AWCRT improvement – up to 50%. These generally modest system-level AWCRT improvements

[9]To quantify the improvement of backtracking, the absolute difference between the AWCRT and the AWCRT with backtracking was normalized to the optimal AWCRT and used to measure the improvement rendered through backtracking

mask the task-level benefits. We examine a system of 20 tasks which saw only a 0.98% improvement in AWCRT through backtracking (98% of systems see an improvement under 1%). The WCRTs of the individual tasks are plotted in figure 5.8. Remarkably, some task WCRTs improved by more than 50%. Hence, on the task level, backtracking the MPTAA can indeed be quite beneficial.

6. Conclusions and Future Work

In this paper we investigate using preemption threshold scheduling to minimize stack size. We present a framework for using PTS for hard real-time systems using both static and dynamic priority allocation schemes. We demonstrate the value and memory optimality of PTS and investigate its performance and also present an improvement.

Open issues remain to be explored. Is there another metric for measuring response time which is more appropriate than the AWCRT? Does minimizing AWCRT also minimize average response time, assuming representative distributions?

References

[1] *The Paparazzi Project*. http://www.recherche.enac.-fr/paparazzi/.

[2] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[3] T. P. Baker. Stack-based scheduling for real-time processes. *Real-Time Syst.*, 3(1):67–99, 1991.

[4] L. Barello. *AvrX Real-Time Kernel*. http://barello-.net/avrx/, 2006.

[5] R. Barry. *Free Real-Time Operating System (FreeRTOS)*. http://www.freertos.org/, 2006.

[6] J. Chen, A. Harji, and P. Buhr. Solution space for fixed-priority with preemption threshold. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 385–394, San Francisco, CA, 2005. IEEE Computer Society.

[7] W. Chen, Z. Wu, and X. Wang. Minimizing memory utilization of task sets in SmartOSEK. *19th International Conference on Advanced Information Networking and Applications*, 02:552–558, 2005.

[8] A. Coombs. Designing an efficient RTOS for a resource-constrained 8-bit microprocessor. Technical report, LiveDevices (Realogy), 2001.

[9] R. Davis, N. Merriam, and N. Trace. How embedded applications using an RTOS can stay within on-chip memory limits. In *In 12th Proceedings of Euromicro Conference on Real-Time Systems*, Maastricht, Netherlands, June 2000.

[10] Express Logic Corporation, http://www.express-logic.com. *ThreadX User Guide*, 2003.

[11] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, London, UK, 2001.

[12] D.-Z. He, F.-Y. Wang, W. Li, and X.-W. Zhang. Hybrid earliest deadline first/preemption threshold scheduling for real-time systems. In *International Conference on Machine Learning and Cybernetics*, 2004.

[13] S. Kim, S. Hong, and T.-H. Kim. Perfecting preemption threshold scheduling for object-oriented real-time system design: from the perspective of real-time synchronization. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 223–232, Berlin, Germany, 2002. ACM Press.

[14] W. Kim, J. Kim, and S. L. Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *Proceedings of the 2004 international symposium on Low power electronics and design (ISLPED'04)*, pages 393–398, Newport Beach, California, USA, 2004. ACM Press.

[15] J. J. Labrosse. *Microc/OS-II*. CMP Books, 2nd edition, 2002.

[16] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *11th Real-Time Systems Symposiom (RTSS' 90)*, Lake Buena Vista, FL, 1990.

[17] J. W.-S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[18] Mentor Graphics Corp., http://www.mentor.com. *Nucleus OSEK*.

[19] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel. Papabench: a free real-time benchmark. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dresden, Germany, 2006.

[20] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Austin, TX, 2002.

[21] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *Trans. on Embedded Computing Sys.*, 4(4):751–778, 2005.

[22] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-timeobject-oriented models. In *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport, CA, USA, 2002.

[23] M. Saksena and Y. Wang. Scalable multi-tasking using preemption thresholds. In *The 6th IEEE Real-Time Technology and Application Symposium*, Cheju Island, South Korea, 2000.

[24] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[25] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption thresholds. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, Hong Kong, China, 1999.