Determining Maximum Stack Usage in Preemptive Shared Stack Systems

Kaj Hänninen[1,2], Jukka Mäki-Turja[1], Markus Bohlin[1,4], Jan Carlson[1] and Mikael Nolin[1,3]
[1]Mälardalen Real-Time Research Centre (MRTC), Västerås, Sweden
[2]Arcticus Systems, Järfälla, Sweden
[3]CC Systems, Uppsala, Sweden
[4]Swedish Institute of Computer Science, Kista, Sweden
kaj.hanninen@mdh.se

Abstract

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties.

We also show how to safely approximate the exact stack usage by using static (compile time) information about the system model and the underlying run-time system on a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system.

Comprehensive evaluations show that our technique significantly reduces the amount of stack memory needed compared to existing analysis techniques. For typical task sets a decrease in the order of 70% is typical.

1. Introduction

In conventional multitasking systems, each thread of execution (task) has its own allocated execution stack. In systems with a large number of tasks a large number of stacks are required. Hence the total amount of RAM needed for the stacks can grow exceedingly large.

Stack sharing is a memory model in which several tasks share one common run-time stack. It has been shown that stack sharing can result in memory savings [9, 16] compared to the conventional stack model. The shared stack model is applicable to both non-preemptive as well as preemptive systems, and it is especially suitable in resource constrained embedded real-time systems with limited amount of memory. Stack sharing is currently supported by many commercial real-time kernels, e.g. [20, 18, 3, 33].

The traditional method to calculate the memory requirements for a shared run-time stack in preemptive systems is to sum the maximum stack usage of tasks in each preemption level and possibly consider additional overheads such as memory used by interrupts and context switches. A major drawback with the traditional calculation method is that it often results in over allocation of stack memory by presuming that all tasks with maximum stack usage in each priority level can preempt each other in a nested fashion during run-time. However, there may, in many cases, be no actual possibility for these tasks to preempt each other (e.g. due to explicit or implicit separation in time). Moreover, the possible preemptions may not be able to occur in a nested fashion.

Taking advantage of the fact that many real-time system exhibit a predictable temporal behavior, it is possible to identify feasible preemption scenarios, i.e., which preemptions can in fact occur, and whether they can occur in a nested fashion or not. Therefore, a more accurate stack analysis can be made. One example of a system that lends itself to such analysis is a hybrid, statically and dynamically, scheduled system. Such a system consists of an off-line scheduler producing the static schedule and a fixed priority scheduler (FPS) that dispatches tasks at run-time. The commercial operating system, Rubus OS by Arcticus Systems AB [3], supports such a system model. The Rubus OS is mainly used in resource-constrained embedded real-time systems. For instance, in the vehicular industry, Volvo Construction Equipment (VCE) [34], BAE Systems Hägglunds [15], and Haldex Traction Systems [13] all use the Rubus OS in their vehicles or components.

In this paper, we present the general problem of analyzing a shared system stack for resource constrained preemptive real-time systems. We provide a general and exact problem formulation applicable for preemptive systems based on dynamic run-time properties. We also present an approximate stack analysis method to derive a safe upper bound on stack usage in static offset based, fixed priority and preemptive systems that use a shared stack. We evaluate and show that the proposed method gives significantly lower upper bounds on stack memory requirements than existing stack dimensioning methods for fixed priority systems.

Paper outline. Section 2 describes related work and sets the context for the contributions of this paper. In sections 3, 4, and 5 we present the exact formulation of determining the maximum stack usage and our safe approximation of the stack usage for our target system model. Section 6 presents an evaluation of our approximative analysis method, and Section 7 concludes the paper.

2. Related work

The notion of shared stack has been used in several publications to describe the ability to utilize either a common run-time stack or a pool of run-time stacks. For example, in [21], stack sharing is performed by having a pool of available stack areas. When a task starts executing, it fetches a stack from the pool, and returns it at termination. In [22], Middha *et al.* address stack sharing in the sense that the stack of a task can grow into the stack area of another task.

In this paper, we use the notion of stack sharing when several tasks use one common, statically allocated, run-time stack. This type of stack sharing can be efficiently implemented in systems where tasks have run-to-completion semantics, and do not suspend themselves. This type of stack sharing is supported by several commercial real-time operating systems, e.g. [18, 3, 33].

2.1. Stack analysis

In [4], Baker presents the Stack Resource Policy (SRP) that permits stack sharing among processes in static and in some dynamic priority preemptive systems. The basic method to determine the maximum amount of stack usage in SRP is to identify the maximum stack usage for tasks at each priority level (or preemption level) and then to sum up these maximums for each priority level. A safe upper bound (SPL) on the total stack usage using information about priority levels can formally be expressed as:

$$SPL = \sum_{l \in \textit{prio-levels}} \max_{i \in \textit{tasks with prio } l} (S_i) \qquad (1)$$

where S_i is the maximum stack usage of task i.

Gai *et al.* [11] present SRP with preemption thresholds (SRPT). They present a procedure to minimize shared stack usage, without jeopardizing schedulability, by use of non-preemption groups for tasks using SRPT. They extend the work of Saksena and Wang [28] by taking the stack usage of tasks into account when establishing non-preemption groups.

In [9] Davis *et al.* address stack memory requirements by using non-preemption groups to reduce the amount of memory needed for a shared stack. They show that the number of preemption levels required for typical systems can be relatively small, while maintaining schedulability.

Although non-preemption groups can reduce the amount of RAM needed for a shared stack, the use of non-preemption groups affects a system by restricting the occurrences of preemptions, which can have a negative affect on schedulability. Also, the method we present in this paper can further reduce the system stack by performing our analysis after preemption groups have been assigned.

2.2. Preemption analysis

A large number of publications address preemption analysis for different reasons, see, e.g. [2, 7, 10, 17, 25, 26, 30]. For example, in [17] Lee *et al.* present a technique to bound cache-related preemption delays in fixed-priority preemptive systems. They account for task phasing and nested preemption patterns among tasks to establish an upper bound on the cache timing delay introduced by preemptions. Our work relates to theirs in the sense that we also investigate occurrences of nested preemption patterns. However, our objectives differ in that Lee *et al.* are mainly interested in timing delays caused by cache reloading and preemption patterns whereas we address shared memory requirements as an effect of nested preemption patterns.

In [10], Dobrin and Fohler present a method to reduce the number of preemptions in fixed priority based systems. They define three fundamental conditions that have to be satisfied in order for a preemption to occur. The same conditions form the basis of our upper bound method described in Section 5.

3. Stack analysis of preemptive systems

The primary purpose of an execution stack is to store local data which consists of variables and state registers, parameters to subroutines and return addresses. Real-time systems typically have a separate stack, statically allocated, for each task. However, under certain conditions, tasks can share stack to achieve a lower overall memory footprint of the system.

In this paper we consider systems where a subset of tasks use a common, statically allocated, run-time stack. For this to be possible, we assume that a task only uses the stack between the start time of an instance and the finishing time of that instance, i.e., no data remains on the stack from one instance of a task to the next. Furthermore, we require non-interleaving task execution [4, 9]. If υ_j begins executing between the start and finish of υ_i, then υ_i is not allowed to resume execution until υ_j has finished. In practice, this is ensured by not allowing tasks to suspend themselves voluntarily, or to be suspended by blocking once they have started their execution. In practice this means that OS-primitives like `sleep()` and `wait_for_event()` cannot be used, and that any blocking on shared resources must be handled before execution start, e.g., with a semaphore protocol like immediate inheritance protocol [6].

We formally define the start and finishing time of a task instance υ_i, as follows:

st_i The absolute time when υ_i actually begins executing.

ft_i The absolute time when υ_i terminates its execution.

At any given point in time, the worst case total stack usage of the system equals the sum of the stack usage for each individual task instance. Thus, with $s_i(t)$ denoting the actual stack usage of υ_i at time t, the maximum stack usage of the system can be expressed as follows:

$$\max_{t \in \textit{time instant}} \sum_{\upsilon_i \in \textit{task instances}} s_i(t) \tag{2}$$

This corresponds to the amount of memory that must be statically allocated for the shared stack to ensure the absence of stack overflow errors. For some systems, e.g., non-preemptive, statically scheduled systems with simple task code, it might be possible to directly compute or estimate $s_i(t)$. In general, however, they are not directly computable before the system is executed.

We note that the total stack usage depends on three basic properties:

(i) the stack memory usage of each task instance

(ii) the possible preemptions that may occur between any two instances

(iii) the ways in which preemptions can be nested

Determining the stack memory usage of a single task instance requires knowledge of the possible control-flow paths within the task code [14]. In [5] Brylow *et al.* present a static checker for interrupt driven software. The checker is able to calculate the stack size of assembler programs by producing a control-flow graph annotated with information about time, space, safety and liveness.

However, due to the difficulties in determining the exact stack usage at every point in time for a given task instance, shared-stack analysis methods often assume that whenever a task is preempted, it is preempted when it uses its maximum stack depth. We make the same assumption, and use S_i to denote the maximum stack usage for task instance υ_i. Thus, when υ_i and υ_j are instances of the same task, we have $S_i = S_j$. Bounds on maximum stack usage for a given task can be derived by abstract interpretation using tools such as AbsInt [1] and Bound-T [31].

In order to calculate the maximum stack usage of the full system, we need to account for all possible preemption patterns. Under the assumption of non-interleaving task execution, a task instance, υ_i, is preempted by another task instance, υ_j, if (and only if) the following holds:

$$st_i < st_j < ft_i \tag{3}$$

In particular, we are interested in chains of nested preemptions. We define a *preemption chain* to be a set $\{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ of task instances such that

$$st_1 < st_2 < \cdots < st_k < ft_k < ft_{k-1} < \cdots < ft_1 \tag{4}$$

Under the assumption that the worst case stack usage of a task occur when the task is preempted, the worst case stack usage SWC for a shared stack preemptive system can be expressed as follows:

$$SWC = \max_{PC \in \textit{preemption chains}} \sum_{\upsilon_i \in PC} S_i \tag{5}$$

This formulation, however, cannot be directly used for analyzing and dimensioning the shared system stack since it is based on the dynamic (only available at run-time) properties st_i and ft_i. To be able to statically analyze the system, one has to relate the static (compile-time) properties to these dynamic properties. This is done by establishing how the system model, scheduling policy, and run-time mechanism constrain the values of the actual start and finishing times. The problem can be viewed as a scheduling problem with the objective of maximizing the total stack usage of the schedule, subject to system constraints on how tasks are ordered in the schedule.

4. System model for hybrid scheduled systems

The system model we adopt is based on the commercial operating system Rubus OS by Arcticus Systems AB [3], which supports the execution of both time triggered and event triggered tasks. The Rubus OS is mainly intended for, and used in dependable resource-constrained embedded real-time systems.

The system model is a hybrid, static and dynamic, scheduled system where a subset of the tasks are dispatched by a static cyclic scheduler (time triggered tasks). The rest of the tasks are dispatched by events in the system (event triggered tasks). The static schedule is constructed off-line and a fixed priority scheduler (FPS) dispatches tasks at run-time. The event-triggered tasks can be categorized in two different classes: (i) event-triggered interrupts which have higher priority than the time-triggered tasks, and (ii) background scheduled event-triggered tasks which have lower priority than the time-triggered tasks.

The time triggered tasks share a common system stack. It is the objective of this paper to analyze, and ultimately dimension this shared system stack efficiently. The time-triggered subsystem is used to host safety critical applications. Hence, to isolate it from any erroneous event-triggered tasks, it uses its own stack.

4.1. Formal system model

The system model used in this paper can be seen as an offset based model with static offsets [12, 23, 24, 32], defined as follows: The system, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i. For non-periodic, events T_i denotes the minimum inter-arrival time between two consecutive events. The activating events are mutually independent, i.e., phasing between them is arbitrary.

A transaction, Γ_i, contains $|\Gamma_i|$ tasks, and each task may not be activated (released for execution) until a time, offset, elapses after the arrival of the activating event.

We use τ_{ij} to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within the transaction. A task, τ_{ij}, is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). Furthermore, S_{ij} is used to denote the maximum stack usage of τ_{ij}. The system model is formally expressed as:

$$\begin{aligned}\Gamma &:= \{\langle \Gamma_1, T_1\rangle, \ldots, \langle \Gamma_k, T_k\rangle\}\\ \Gamma_i &:= \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\}\\ \tau_{ij} &:= \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij}, S_{ij}\rangle\end{aligned}$$

We assume that the system is schedulable and that the worst case response-time for each task, (R_{ij}), has been calculated [24].

Due to the non-interleaving criterion for stack sharing, we require that tasks exhibit run-to-completion semantics when activated, i.e., they cannot suspend themselves. An invocation of a task can be viewed as a function call from the operating system, and the invocation terminates when the function call returns. When tasks share the same priority, they are served on a first-come first-served basis.

We assume that if access to shared resources is not handled by the static scheduler by time separation, a resource sharing protocol where blocking is done before start of execution is employed (such as the stack resource protocol [4] or the immediate inheritance protocol [6]).

Relating back to Rubus OS, one can view the system as a transaction based system with one transaction, Γ_t, corresponding to the static schedule (time-triggered tasks) and any number of transactions corresponding to higher priority event triggered tasks (interrupts). For the even-triggered transactions there are no restrictions placed on offset, deadline or jitter, i.e., they can each be either smaller or greater than the period. Since Γ_t represents the static schedule, which is cyclical with period T_t, offset, jitter and deadline are less than the period, i.e., $O_{tj}, D_{tj}, J_{tj} \leq T_t$ for the time-triggered transaction. How a scheduler can generate a feasible schedule with interfering interrupts is described in [29, 23].

It is the objective of this paper to find a tight upper bound on the shared system stack for the tasks in the time-triggered transaction Γ_t. Task j belonging to Γ_t we will denote τ_{tj}. The tasks in the transaction can be preempted by other tasks in the transaction and by higher priority event triggered tasks.

5. Stack analysis of hybrid scheduled systems

In this section, we describe a polynomial time method to establish a safe upper bound on the shared stack usage for the system model described in Section 4. The upper bound is safe in the sense that the run-time stack can never exceed the calculated upper bound.

A safe upper-bound estimate of the exact problem can be found by using offsets and maximum response times as approximations of actual start and finishing times. Generalizing the preemption criteria identified by Dobrin and Fohler [10], we form the binary relation $\tau_{ti} \prec \tau_{tj}$ with the interpretation that τ_{ti} may be preempted by τ_{tj}. The relation holds whenever (1) τ_{ti} can become ready before τ_{tj}, (2) τ_{ti} possibly finishes (i.e., has a response time) after the start of τ_{tj}, and (3) τ_{ti} has lower priority than τ_{tj}. The relation can now formally be defined as:

$$\tau_{ti} \prec \tau_{tj} \equiv O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti} \wedge P_{ti} < P_{tj} \tag{6}$$

Lemma 1 *The $\prec$ relation is a safe approximation of the possible preemptions between tasks in Γ_t. That is, if τ_{ti} can under any run-time circumstance be preempted by τ_{tj}, then $\tau_{ti} \prec \tau_{tj}$ will hold.*

Proof of Lemma 1 *Suppose that τ_{ti} is preempted by τ_{tj}. We show that this implies (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, (2) $O_{tj} < R_{ti}$, and (3) $P_{ti} < P_{tj}$.*

(3) follows directly from the preemption. Now let t be the time instant when τ_{tj} has finished blocking, which implies $t \leq O_{tj} + J_{tj} + B_{tj}$. Then a possibly empty interval $[t, st_{tj}]$ of execution with higher priority than τ_{tj} follows, in which τ_{ti} cannot execute because $P_{ti} < P_{tj}$. Since τ_{ti} must start before τ_{tj}, we can conclude that $st_{ti} < t$, which together with $O_{ti} \leq st_{ti}$ and $t \leq O_{tj} + J_{tj} + B_{tj}$ gives us $O_{ti} < O_{tj} + J_{tj} + B_{tj}$ and (1). From Equation 3 we have $st_{tj} < ft_{ti}$ and this together with $O_{tj} \leq st_{tj}$ and $ft_{ti} \leq R_{ti}$ leads to $O_{tj} < R_{ti}$ and (2), which completes the proof. $\square$

The upper-bound problem can now be informally stated as finding the maximum stack usage of all possible preemption chains in Γ_t. This equals finding the time instant in the schedule which has a maximum stack usage, given the approximation of actual start and finishing times with offsets and response times respectively, and assuming that at all preemptions the preempted task uses its maximal stack.

A sequence Q of tasks is a *possible preemption chain* (PPC) if it holds that $\tau_{ti} \prec \tau_{tj}$ for all τ_{ti}, τ_{tj} in Q where τ_{ti} occurs before τ_{tj} in the sequence. The stack usage SU_Q of a PPC Q is the sum of the stack usage of the individual tasks in the chain, i.e., $SU_Q = \sum_{\tau_{ti} \in Q} S_{ti}$.

A straightforward computation of a safe upper bound for a set of tasks involves computing the stack usage for all PPCs. However, for a set of n tasks there exist $2^n - 1$ different PPCs in the worst case, which yields an exponential time complexity for an algorithm based on this idea. A more efficient algorithm can be constructed by first finding sets of tasks which all overlap in time without regarding priorities. These sets can then be investigated, in turn, to find a PPC with maximal stack usage.

We let the relation $\tau_{ti} \preceq \tau_{tj}$ hold whenever the semi-closed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$ intersect, or more formally:

$$\tau_{ti} \preceq \tau_{tj} \equiv O_{ti} < R_{tj} \wedge O_{tj} < R_{ti} \tag{7}$$

The relation $\preceq$ is a relaxation of the $\prec$ relation. That is, $\tau_{ti} \prec \tau_{tj} \rightarrow \tau_{ti} \preceq \tau_{tj}$. To see this, suppose that $\tau_{ti} \prec \tau_{tj}$ which implies $O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti}$, according to Equation 6. Since $O_{tj} + J_{tj} + B_{tj} \leq R_{tj}$ follows from the notion of response time, we have $O_{ti} < R_{tj} \wedge O_{tj} < R_{ti}$, which also is the definition of $\tau_{ti} \preceq \tau_{tj}$.

We can now define an *overlap set* K_r as a set of tasks where:

$$\forall \tau_{ti}, \tau_{tj} \in K_r : \tau_{ti} \preceq \tau_{tj}$$

The stack usage SU_{K_r} of an overlap set K_r is defined as the maximum stack usage SU_Q of all PPCs Q where $Q \subseteq K_r$:

$$SU_{K_r} = \max_{\forall Q \subseteq K_r : PPC(Q)} (SU_Q) \tag{8}$$

K_r is *maximal*, if and only if, there exists no overlap set, K_s, such that $K_r \subset K_s$.

Lemma 2 *For any PPC Q, there exists a maximal overlap set K_r such that $Q \subseteq K_r$.*

Proof of Lemma 2 *From the definitions of a PPC and the $\prec$ and $\preceq$ relations, we know that for all tasks $\tau_{ti} \prec \tau_{tj}$ in Q it also holds that $\tau_{ti} \preceq \tau_{tj}$, and thus Q is an overlap set. Then, either Q is maximal, or it can become maximal by extending it with additional tasks. In either case, the lemma holds.* $\square$

In all, the algorithm for computing the upper bound SUB on the maximum stack usage for a set of tasks Γ_t can be summarized as follows:

1. Find the maximal overlap sets in Γ_t:
   $K = \{K_1, K_2, \ldots, K_k\}$.
2. For each of them, compute SU_{K_r} according to Equation 8.
3. The upper bound of the stack usage for Γ_t can now be computed as follows:

$$SUB = \max_{\forall K_r \in K} (SU_{K_r}) \tag{9}$$

Informally, we start by finding all sets of tasks that can overlap in time based on their offsets and worst case response times, which safely approximates their actual start and finishing times. For each such set (K_i), we find all possible preemption chains (PPCs) by also taking task priorities and maximal jitter and blocking time into account, and compute the stack usage for each chain. The stack usage of K_i is the maximum stack usage of all its PPCs, and the maximum stack usage (SUB) of the system is then obtained by taking the maximum stack usage of every K_i.

5.1. Correctness

In order to claim correctness of our approximate stack analysis method, we have to show that it never underestimates the actual stack usage that can occur during run-time.

Theorem 1 *The value computed by the SUB algorithm is a safe upper bound on the actual worst case stack usage for tasks in Γ_t. Formally, $SWC \leq SUB$.*

Proof of Theorem 1 *Let $\Psi \subseteq \Gamma_t$ be the sequence of tasks instances participating in the preemption situation which cause the worst case stack usage, that is, $SWC = \sum_{\tau_{ti} \in \Psi} S_{ti}$. According to Lemma 1, we must have $\tau_{ti} \prec \tau_{tj}$ for tasks τ_{ti} and τ_{tj} that occur in that order in Ψ, and thus Ψ is a PPC with $SU_\Psi = SWC$. Then, Lemma 2 ensures that there exists a maximal overlap set K_r such that $\Psi \subseteq K_r$, and we have $SU_\Psi \leq SU_{K_r}$. Thus, $SWC \leq SU_{K_r} \leq SUB$, which concludes the proof.* $\square$

5.2. Computational complexity

The relaxation of $\prec$ into interval intersection (Equation 7) allows us to efficiently compute an upper bound on the stack usage (Equation 9) by applying a polynomial longest path algorithm on the linearly-bounded number of maximal overlap sets.

To first see that the set of maximal overlap sets $K = \{K_1, K_2, \ldots, K_k\}$ contain at most n elements, i.e., $k \leq n$, consider the graph (Γ_t, E), where Γ_t is the set of vertices and $E = \{\tau_{ti}\tau_{tj} \mid (\tau_{ti} \preceq \tau_{tj}) \wedge \tau_{ti}, \tau_{tj} \in \Gamma_t\}$ is the set of edges. From Equation 7, we have that edges $\tau_{ti}\tau_{tj} \in E$ correspond to intersection of the semi-closed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$, and therefore the graph is an *interval graph* [19]. Because every interval graph is also *chordal* [19], all maximal complete subgraphs in (Γ_t, E), which correspond to all maximal overlap sets, can be found in linear time [27]. Furthermore, for chordal graphs there exists at most n such sets, and thus we have at most n overlap sets [19].

The problem of finding the worst PPC within a single overlap set K_r is significantly easier than for an arbitrary set of tasks. Since it holds that $\tau_{ti} \preceq \tau_{tj}$ for all tasks $\tau_{ti}, \tau_{tj} \in K_r$, and therefore in particular that $O_{ti} < R_{tj}$ for all tasks in K_r, we need only look for a maximum stack usage chain Q where (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, and (2) $P_{ti} < P_{tj}$ for all tasks τ_{ti} and τ_{tj} in that order in Q to find the worst PPC. A directed graph consisting of tasks in K_r and arcs corresponding to properties (1) and (2) is acyclic, and for such graphs a longest-path type algorithm can be used to find the worst PPC [8]. There exist longest-path algorithms with a time complexity of $O(n+m)$, where n is the number of tasks and m is the number of possible preemptions, of which there are at most $n(n-1)/2$. Taking the maximum of a maximal PPC in each set, K_r, of which there are at most n, we will, therefore, find a maximum stack size PPC in at most $O(n^3)$ time.

6. Evaluation

We evaluate the efficiency of our proposed method to establish a safe upper bound on shared stack usage by randomly generating realistic sized task sets. The size, load and stack usage of the task sets are derived from a wheel-loader application by Volvo Construction Equipment [34]. We use three different methods to calculate the shared system stack usage:

SPL	The traditional method to dimension a shared system stack by summing up the maximum stack usage in each priority level.
SUB	The safe upper bound on the shared stack usage presented in Section 5
SLB	A lower bound on on the shared stack usage, for each task set.

The lower bound is established using simple heuristics that tries to maximize shared stack usage by generating only feasible preemption scenarios for the task set, and thus, represents scenarios that definitely can occur. From all PPCs, the heuristic selects a sample set of roughly 500 chains. For each of them, the method tries to construct a feasible arrival pattern for the ET tasks and actual execution time values that cause an actual preemption between the tasks in the chain. The quality of this heuristic method degrades as the length of the chains or the total number of PPCs increases, which can be seen in the figures.

By establishing a safe upper bound and a feasible lower bound, we know that the actual worst case stack usage is bounded by SUB and SLB. The reason for including SLB is to give an indication on the maximum amount of improvement there might be for SUB.

6.1. Simulation setup

In our simulator we generate random task sets as input to the stack analysis application. The task generator takes the following input parameters:

- Total number of TT (time triggered) tasks (default = 250)
- Total load of TT tasks (default = 60%)
- Minimum and maximum priorities of TT tasks (default = 1 and 32)
- Minimum and maximum stack usage of TT tasks (default = 128 and 2048)
- Total number of ET (event triggered) tasks (default = 8)
- Total load of ET tasks (default = 20%)
- The shortest possible minimum inter-arrival time of an ET task (default = 1.000)

The generated schedule for TT tasks is always 10.000 time units. All ET tasks have higher priority than TT tasks. The default values for the input parameters represent a base configuration derived from a real application [34].

Using these parameters a task set with the following characteristics is generated:

- Each TT offset (O_{ti}) is randomly and uniformly distributed between 0 and 10.000.
- Worst case execution times for TT tasks, C_{ti}, are initially randomly assigned between 1 and 1000 time units. The execution times get adjusted by multiplying all C_{ti} by a fraction, so that the the TT load (as defined by the input parameter) is obtained.
- TT priorities are assigned randomly between minimum and maximum value with a uniform distribution.

6.2. Results

Each diagram shows three graphs corresponding to the stack usage calculated by the three methods: SPL, SUB, and SLB. Each point in the graphs represents the mean value of 100 generated task sets. We also measured the 95% confidence interval for the mean values. These are not shown because of their small size (less than 7% of the y-value for each point). We also measured the CPU time to calculate an upper bound on shared stack usage for each generated task

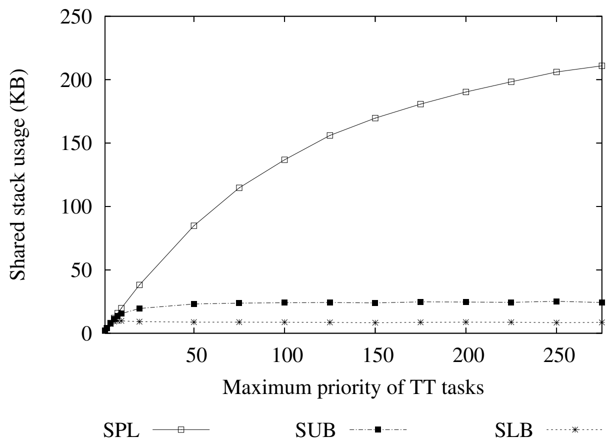

Figure 1. Varying the number of priority levels of TT tasks

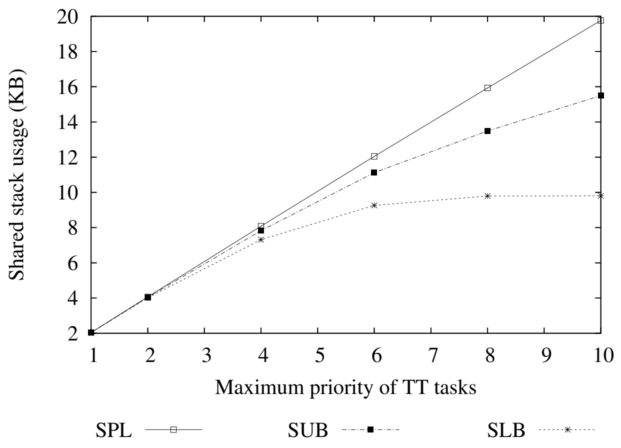

Figure 2. Varying the number of priority levels of TT tasks (zoom of Fig. 1)

set. Using the method described in Section 5, the calculations took less than 63ms per task set, on an Intel Pentium 4, 2.8GHz with 512MB of RAM.

In Fig. 1, we vary the maximum priority for TT tasks between 1 and 300, keeping the minimum priority at 1. This gives a distribution of possible priorities (priority levels), from 1 to n, where n is indicated by the x-axis. We see, in Fig. 2 which zooms in on Fig. 1, for maximum priorities up to 10, that the difference in stack usage between SPL and SUB is less noticeable with a low number of priority levels. However, for larger numbers of priority levels the difference is significant. SPL is not expected to flatten out before all tasks actually have unique priorities, whereas our method (SUB) flattens out significantly earlier. We conclude that the maximum number of tasks in any preemption chain is increasing very slowly (or not at all) when the number of TT tasks increases above a certain value, since the system load is constant.

In Fig. 3, we vary the maximum stack usage of each TT task between 128 bytes and 4096 bytes. We do this by assigning an initial stack of 128 bytes for each TT task, i.e. initially the stack size variation is zero. We then vary the stack size between 128 and 512 bytes, 128 and 1024 bytes, and so on. The diagram shows that SUB gives significantly lower values on shared stack usage than the traditional SPL. We also notice that an increase in stack variation scales up, linearly, the differences between SPL and SUB. The linearity is expected, since an increase in stack variation does not affect occurrences of possible preemptions in the system, i.e., possible nested preemptions are retained.

In Fig. 4 we vary the maximum number of TT tasks between 5 and 275. We see that the shared stack usage of the traditional SPL is dramatically increasing in the beginning. This is due to the fact that when the number of TT

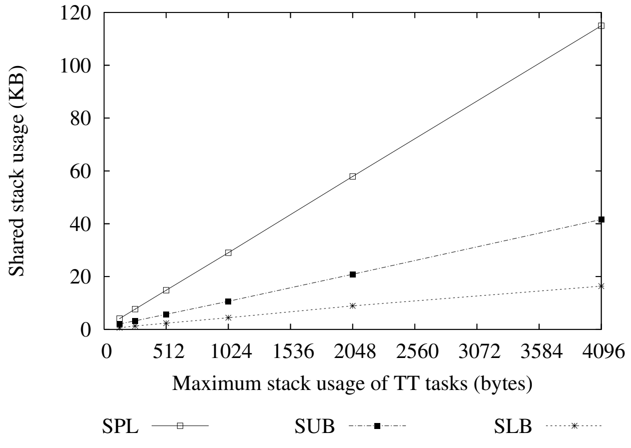

Figure 3. Varying stack usage of TT tasks

tasks is lower than the maximum priority of TT tasks (32), most TT tasks have unique priorities. SUB, on the other hand, increases much slower than SPL because the maximum number of tasks involved in any preemption chain is slowly increasing. SUB is expected to further approach SPL since increasing the number of tasks will increase the likelihood of larger number of tasks involved in the preemption chains.

In Fig. 5, we vary the total load of TT tasks between 10% (0.1) and 70% (0.7). The figure shows that the shared stack usage of SPL is constant, whereas, SUB is slowly increasing. SPL is expected to be constant since it is only affected by the number of priority levels and unaffected by the actual preemptions that can occur in a system. The increase of SUB is due to increasing response-times of TT tasks when the TT load increases, which will increase the likelihood of larger number of tasks involved in nested preemptions.

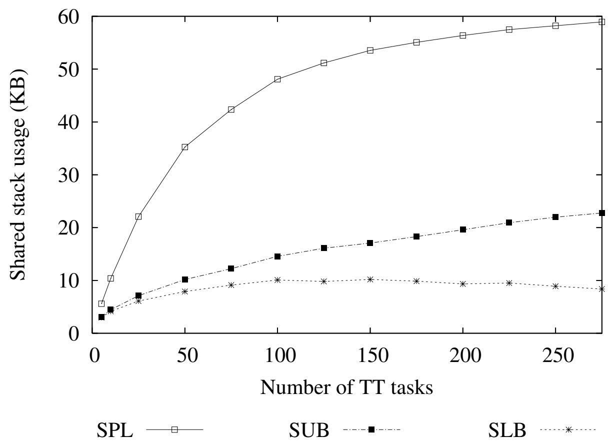

Figure 4. Varying the number of TT tasks

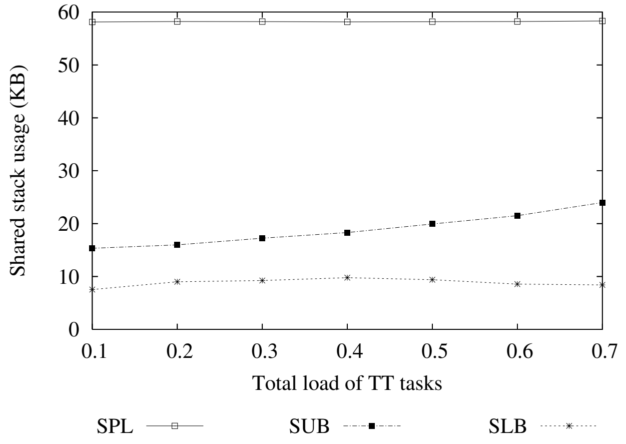

Figure 5. Varying the load of TT tasks

7. Conclusions and future work

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties.

By approximating these run-time properties, together with information about the underlying run-time system, we present a method to safely approximate the maximum system stack usage at compile time. We do this for a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system. Such a system model provides lot of static information that we can use to estimate the dynamic start- and finishing-times. Our method finds the nested preemption pattern that results in the maximum shared stack usage. We prove that our method is a safe upper bound of the exact system stack usage and show that our method has a polynomial time complexity.

In a comprehensive simulation study, we evaluated our technique and compared it to the traditional method to estimate stack usage. We find that our method significantly reduces the amount of stack memory needed. For realistically sized task sets, a decrease in the order of 70% is typical.

In this paper, we focused on a system model for a given commercial real-time operating system. In the future, we plan to extend our approximation method to a more general system model, to incorporate all the features of the general model for tasks with offsets [12]. Such an extension would make the presented analysis technique applicable to a wider range of systems.

Our current method could also be extended to account for other types of information that can further limit the number of possible preemptions. We currently only account for separation in time (offsets and response-times) between tasks. However, in many systems other types of information, such as precedence and mutual-exclusion relations may exists between tasks, thus limiting the possible preemptions.

The method presented here could also be used in synthesis and configuration tools that generate optimized systems from given application constraint. In this case, the results from our analysis can be used to guide optimization or heuristic techniques that try to map application functionality to run-time objects.

References

[1] Absint. Web page, http://www.absint.com/stackanalyzer/.

[2] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computing Systems*, 15(2):134–165, May 1997.

[3] Arcticus systems. Web page, http://www.arcticus-systems.se.

[4] T. P. Baker. A stack based resource allocation policy for real-time processes. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990.

[5] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[6] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996.

[7] H. Cho, B. Ravindran, and E. D. Jensen. A space-optimal wait-free real-time synchronization protocol. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, July 2005.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2001.

[9] R. Davis, N. Merriam, and N. Tracey. How embedded applications using an RTOS can stay within on-chip memory limits. In *Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[10] R. Dobrin and G. Fohler. Reducing the number of preemptions in fixed priority scheduling. In *16th Euromicro Conference on Real-time Systems*, Catania, Sicily, Italy, July 2004.

[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 *Proceedings of the 22nd Real-Time Systems Symposium*, London, UK, Dec 2001.

[12] J. C. P. Gutierrez and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th Real-Time Systems Symposium*, Dec 1998.

[13] Haldex traction systems. Web page, http://www.haldex-traction.com/.

[14] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Proceedings of the Design, Automation and Test in Europe*, March 2005.

[15] BAE Systems Hägglunds. Web page, http://www.haggve.se.

[16] K. Hänninen, J. Lundbäck, K.-L. Lundbäck, J. Mäki-Turja, and M. Nolin. Efficient event-triggered tasks in an RTOS. In *Proceedings of the International Conference on Embedded Systems and Applications*, June 2005.

[17] C. G. Lee, K. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, Sept 2001.

[18] Live devices etas group. Web page, http://en.etasgroup.com/products/rta/.

[19] T. A. McKee and F. McMorris. *Topics in intersection graph theory*. SIAM Monographs on Discrete Mathematics and Applications #2. Society for Industrial and Applied Mathematics (SIAM), 1999.

[20] Micro digital. Web page, http://www.smxinfo.com/mt.htm.

[21] Micro Digital Inc. *smx Features and Architecture*.

[22] B. Middha, M. Simpson, and R. Barua. MTSS: Multi task stack sharing for embedded systems. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Francisco, CA, Sept 2005.

[23] J. Mäki-Turja, K. Hänninen, and M. Nolin. Efficient Development of Real-Time Systems Using Hybrid Scheduling. In *International conference on Embedded Systems and Applications (ESA)*, June 2005.

[24] J. Mäki-Turja and M. Nolin. Fast and Tight Response-Times for Tasks with Offsets. In *Proc. of the 17^{th} Euromicro Conference on Real-Time Systems*, July 2005.

[25] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2006.

[26] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Dec 2002.

[27] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 245–254, New York, NY, USA, 1975. ACM Press.

[28] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the 21st Real-Time System Symposium*, Nov 2000.

[29] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98), 27-29 October 1998, Hiroshima, Japan*, pages 158–165. IEEE Computer Society, 1998.

[30] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, July 2005.

[31] Tidorum. Web page, http://www.tidorum.fi/bound-t/.

[32] K. Tindell. Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[33] Unicoi systems. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

[34] Volvo Construction Equipment. Web page, http://www.volvoce.com.