Safe Shared Stack Bounds in Systems with Offsets and Precedences

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

Abstract

The paper presents two novel methods to bound the stack memory used in preemptive, shared stack, real-time systems. The first method is based on branch-and-bound search for possible preemption patterns, and the second one approximates the first in polynomial time. The work extends previous methods by considering a more general task-model, in which all tasks can share the same stack. In addition, the new methods account for precedence and offset relations. Thus, the methods give tight bounds for a large set of realistic systems. The methods have been implemented and a comprehensive evaluation, comparing our new methods against each other and against existing methods, is presented. The evaluation shows that our exact method can significantly reduce the amount of stack memory needed. In our simulations, a decrease in the order of 40% was typical, with a runtime in the order of seconds. Our polynomial approximation consequently yields about 20% higher bound than the exact method.

1. Introduction

In order to limit the amount of RAM set aside for stack-memory in embedded systems, many RTOSes provides means to execute multiple tasks on a single, shared, stack (e.g. Rubus [3], Fusion [27], Erika [10], SMX [17], etc.). In order to make maximum use of this ability to share stack-memory we need methods to properly dimension the memory allocated to the stack. This paper shows how to exploit commonly available knowledge of precedence and offsets between tasks to calculate a tight upper bound on the amount of stack-memory used.

In shared stack systems, one stack-frame is added to the system's stack for each level of preemption. Thus, the maximum stack-usage occurs during some worst-case preemption pattern. In simple task models (commonly used in real-time scheduling theory), where tasks are assumed to be independent, any preemption pattern is possible — thus we have to pessimistically assume that all tasks may be active and preempted at the point where they use the most stack. The system's maximum stack-usage thus becomes $\sum S_i$ (where S_i denotes the maximum stack-usage of task i). The consequence is that in these models the benefits of using a shared stack is limited.

In many systems we have information that let us deduce that some preemption patterns are impossible. For example, in a system where multiple tasks share the same priority, no preemptions among these tasks are possible (assuming FIFO scheduling within a priority level and an early-blocking resource allocation protocol such as the immediate inheritance protocol). In this case, the system's maximum stack-usage becomes $\sum_p \max_p(S_i)$ (where p denotes a priority level and $\max_p$ maximizes over the tasks within that priority level). If the number of priority levels is low enough, this type of analysis can provide a much lower bound on stack usage than the above sum over all tasks. Davis *et al.* describes this type of stack analysis and generalize it to allow non-preemption groups to be defined [8].

However, limiting the scheduler by lowering the number of priority levels or manually defining non-preemption groups has drawbacks, since it limits the schedulability of the system and places extra burden on the engineers to define non-preemption groups. Also, in many systems there is even more information available that would allow us to further reduce the possible preemptions in the system.

In this paper we present novel techniques to exploit information about precedence and offset relations between tasks to further limit the number of possible preemption-patterns. We perform a system wide preemption analysis to find the worst case preemption pattern with respect to stack usage. This allows us to calculate a tight bound on the amount of stack memory needed in the system. The intuition behind the techniques is that tasks that have precedence relations will never preempt each other, and tasks

with offset relations may only preempt each other if the response-time of the first task is longer than the offset to the second task. Thus, a prerequisite to perform our analysis is that the response-time and jitter are known for all tasks. We build our analysis on the transactional task-model introduced by Tindell [26] and extended to handle precedences by Gutierrez and Harbour [13]. Given the safe approximations of response-times and jitter resulting from the schedulability analysis presented by, e.g., Mäki-Turja and Nolin [19], we here present two methods to bound the system stack usage. We present one algorithm that searches the whole search space of possible preemptions which has exponential complexity, and a safe approximation method with polynomial complexity. We provide an evaluation of the two methods, comparing them with each other and with the method of summation over priority levels described above.

The transactional task-model allows for modeling of large, complex and realistic real-time systems. Hence, the methods presented have a clear practical value. The methods can be used in a verification/validation phase of system development in order to formally verify that stack overflow will not occur during runtime. The approximation method (due to its better run-time complexity) could also be used in optimizing allocation, mapping, and configuration tools that automate the process of allocating tasks to nodes in distributed systems.

Paper outline. The remainder of this paper is organized as follows. Section 1.1 describes related work and sets the context for the contributions of this paper. In Section 2, we discuss stack sharing and its consequences, and in Section 3 we present the system model that we use. Section 4 presents the exact formulation of determining the maximum stack usage, and gives the theoretical framework for Section 5, which describes algorithms for bounding the stack usage of systems with offsets and precedences. Section 6 gives an experimental evaluation of our analysis methods, and Section 7 concludes the paper and suggests future work.

1.1. Related work

A large number of publications have addressed preemption analysis for specific reasons, see, e.g. [2, 9, 15, 20, 21, 24]. Our work is related in the sense that we also investigate possible preemptions. However, our objectives differ, since we analyze system wide premption patterns to investigate their effect on stack memory requirements for a task model with offsets and precedences.

Throughout the years, a number of publications, have addressed stack sharing. Baker presented the Stack Resource Policy (SRP) that permits stack sharing among processes with shared resources [4]. Chatterjee *et al.* study stack boundedness for interrupt-driven programs [6]. In [8] Davis *et al.* address stack memory requirements and non-preemption groups to reduce shared stack usage. Gai *et al.* [11] present the Stack Resource Policy with preemption Thresholds (SRPT) which extends the work of Saksena and Wang [23] by accounting for stack usage when establishing non-preemption groups. In [12] Ghattas and Dean investigate stack space requirements under preemption threshold scheduling. Middha *et al.* [18] propose the MTSS stack sharing technique that allows a stack to grow into other tasks. In [22] Regehr *et al.* present a method to guarantee stack safety of interrupt-driven software by computing the worst-case memory requirements of individual interrupt handlers and perform preemption analysis between handlers. In [14] we presented an approximate stack analysis method to derive a safe upper bound on the shared stack usage of a static time-driven schedule in offset-based, hybrid scheduled (interrupt- and time-driven) fixed priority preemptive systems. In this paper, we extend that work by supporting stack sharing across several transactions for the task model with offsets. Here we also take precedence relations into account to further reduce possible preemptions.

2. Stack sharing in preemptive systems

In this paper we consider systems where several tasks use a single, statically allocated, run-time stack. For this to be possible task only uses the stack between the start time of an instance, v_i, and the finishing time of that instance, i.e., no data remains on the stack from one instance of a task to the next. This is ensured by not allowing tasks to suspend themselves voluntarily. In practice this means that OS-primitives like `sleep()` and `wait_for_event()` cannot be used. 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 (thus any persistent context must be stored outside of the stack).

It is also required that a task instance never experiences blocking once it has started execution, i.e., we never need to preempt the executing task because a needed resource is locked by a lower priority task. This is achived by using an *early blocking* resource access protocol such as the immediate inheritance protocol [5] or the stack resource policy [4].

The motivation for allowing tasks to share a common stack is that this shared stack can be smaller than the sum of the individual stacks without jeopardizing the correctness of the application. Shared-stack analysis aims at (pre run-time) deriving a safe, but tight, approximation of the worst case (run-time) size of the shared stack. As long as the amount of memory statically allocated for the shared stack exceeds this bound, the absence of stack overflow errors is guaranteed.

At any given point in time, the size of the shared stack equals the sum of the current stack usage for each active

task instance. The maximum size of the shared stack thus depends on two factors: (i) the stack memory usage of each task instance, and (ii) the possible preemption patterns among tasks.

Due to the difficulties in determining the exact stack usage at every point in time for a given task instance, shared-stack analysis methods typically assume that whenever a task is preempted, it is preempted at its maximum stack depth. We make the same assumption. Bounds on maximum stack usage for a given task can be derived by abstract interpretation using tools such as AbsInt [1] and Bound-T [25].

Previous traditional approaches to account for the second factor, i.e., the possible preemption patterns, is based on the fact that at most one task from each priority level (or preemption level, if these two concepts do not coincide) can be active at the same time. Thus, a simple and safe approach for bounding the maximum shared stack usage is to sum the maximum individual stack usage of tasks at each priority (or preemption) level. We call this approach SPL (Sum of all Priority Levels), as desribed by Davis *et al.* [8], and it uses the following function calculate a bound on the stack usage:

$$\sum_{p \in \text{all priority levels}} \max\left(\{S_i : \tau_i \text{ has priority } p\}\right) \quad (1)$$

where S_i denotes the maximum stack usage of task τ_i.

However, this approach can be very pessimistic, since it assumes a worst-case situation where tasks with maximum stack usage from each priority level preempt each other in a nested fashion. In practice, this situation could be impossible to achieve because of factors such as release times, deadlines and precedence constraints that affect when tasks can execute.

The analysis approach proposed in this paper reduces the pessimism of the traditional method by investigating the possible preemption patterns in more detail. 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.

A task instance υ_i is preempted by another task instance υ_j if (and only if) the following holds:

$$st_i < st_j < ft_j < ft_i. \quad (2)$$

Note that the use of an early-blocking resource protocol ensures $ft_j < ft_i$ if $st_i < st_j$.

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

$$st_1 < st_2 < \cdots < st_k < ft_k < ft_{k-1} < \cdots < ft_1. \quad (3)$$

Lemma 1 *$PC = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ is a preemption chain if and only if for all instances υ_i, υ_j in PC where $i < j$, it holds that $st_i < st_j < ft_j < ft_i$.*

Proof of Lemma 1 *The proof of Lemma 1 follows trivially from Equations (2) and (3).*

Let $AllPC$ be the set of all preemption chains in all run-time scenarios. Then, under the assumption that the worst case stack usage S_i of a task instance υ_i can occur at any time during its execution, a bound on the worst case stack usage SWC for a preemptive shared stack system can be expressed as follows:

$$SWC = \max_{PC \in AllPC} \sum_{\upsilon_i \in PC} S_i. \quad (4)$$

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 task 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.

In previous work we have described how this can be done for the special case that only tasks in the same transaction share stack [14]. This paper extends the analysis in the sense that we allow stack sharing among arbitrary interrupt- and/or time-driven transactions consisting of fixed priority tasks with offsets. We also improve the way precedence relations are accounted for in the preemption analysis.

3. System model

The system model used in this paper is an offset-based model [13, 19, 26], defined as follows: the system, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_s is activated by an event, and T_s denotes the minimum inter-arrival time between two consecutive events. The activating events can be mutually independent, i.e. the transactions may execute with arbitrary phasing.

A transaction Γ_s contains $|\Gamma_s|$ tasks. A task may not be released for execution until a certain time (the *offset*) has elapsed after the arrival of the activating event.

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

When referring to the stack usage of a specific instance υ_j of a task τ_{si} we sometimes use S_j instead of S_{si} to simplify the presentation.

The system model is formally expressed as:

$$\begin{aligned}\Gamma :=& \{\langle \Gamma_1, T_1\rangle, \ldots, \langle \Gamma_k, T_k\rangle\}\\ \Gamma_s :=& \{\tau_{s1}, \ldots, \tau_{s|\Gamma_s|}\}\\ \tau_{si} :=& \langle C_{si}, O_{si}, D_{si}, J_{si}, B_{si}, P_{si}, S_{si}\rangle\end{aligned}$$

There are no restrictions placed on deadline or jitter, i.e., they can each be either smaller or greater than the period. We assume that offsets are nonnegative and smaller than the period.

We assume that the system is schedulable and that the worst-case response time for each task, (R_{si}), has been calculated [19].

In addition, we define a binary non-preemption relation NOPRE between tasks such that if NOPRE(τ_{si}, τ_{tj}) then τ_{si} cannot be preempted by τ_{tj}. The relation is reflexive, because two instances of the same task can never interrupt each other. For the analysis in this paper, precedences between tasks in the system are taken into account by encoding these as non-preemption relations, since two tasks with a precedence relation between them will never interrupt each other. Any other mutual exclusion information can, if available, be encoded in the same way.

We assume the system is scheduled with fixed priority scheduling with fifo-scheduling of tasks with the same priority. We assume that an early-blocking resource access protocol, such as the immediate inheritance protocol, is used.

4. Preemption analysis for offset-based systems

In the rest of the paper we assume that all tasks share a common stack. The upper bound problem for multiple transactions can then be informally stated as finding the maximum stack usage of all possible preemption chains, no matter in which transaction they occur.

Stack analysis for multiple transactions is naturally more complex than analysis of one single transaction, since tasks in different transactions may interfere in nontrivial ways depending on relative priorities and the phasing between transactions. To get a safe upper bound on the stack size we therefore need to examine all possible phasing patterns between transactions.

A straightforward approach for analyzing multiple transaction stack behavior is to analyse the transactions in isolation, using the sum for all transactions as an upper bound on the total stack consumption. Each transaction can be analyzed using the method developed in [14] or any other method. The result obtained is a safe upper bound if the analysis for each transaction is safe.

Unfortunately, the latter approach ignores that the global stack upper bound may be significantly lower, since all possible transaction-local preemption patterns may not occur at the same time. One example of this is when two stack-intensive tasks with equal priority both influence the stack bound in their respective transaction. The bound obtained can be pessimistic since no two tasks with equal priority can both be active at the same time.

In this section, we propose a new, more elaborate algorithm which takes this into account. The method is based on a precise analysis of the relaxed global precedence chains that are possible. The algorithm has a non-polynomial time complexity but is nonetheless usable for analyzing realistically sized task sets. However, since sufficiently large task sets will never be analyzable using non-polynomial algorithms, we also propose a less accurate but still competitive approximate method with a polynomial time complexity. The method is a generalization of the one presented in [14] to handle several transactions.

4.1. Pairwise preemptions

Since task preemption is one of the factors influencing the size of the shared stack, a first step is to formulate a safe approximation of possible pairwise preemptions. For this, we first define the release time rt_i of a task instance υ_i as the absolute time when υ_i is ready to start executing.

Let α_k denote the activation time of the transaction releasing an instance υ_k of a task τ_{si}. Then, we have the following relations on the start time and release time of υ_k:

$$\alpha_k + O_{si} \leq st_k. \tag{5}$$

$$rt_k \leq \alpha_k + O_{si} + J_{si}. \tag{6}$$

These concepts are illustrated in Figure 1.

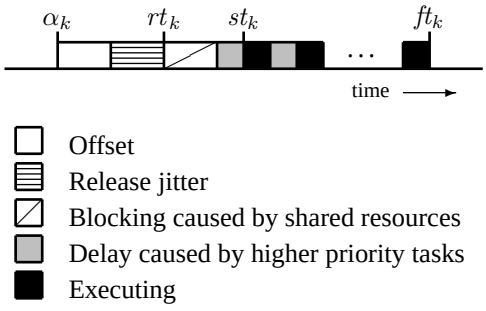

Figure 1. Important activities and time points for a task instance υ_k.

We use ψ_{sji} to denote the offset phasing between two tasks τ_{sj} and τ_{si} in the same transaction Γ_s, and define it

as the minimum distance from an instance of τ_{sj} to the next instance of τ_{si}, or formally:

$$\psi_{sji} = (O_{si} - O_{sj}) \bmod T_s. \tag{7}$$

Generalizing the preemption criteria by Dobrin and Fohler [9], which is further extended in [14], we form the binary relation $\tau_{si} \prec \tau_{tj}$ with the interpretation that τ_{si} may be preempted by τ_{tj}. We let the relation hold whenever (a) τ_{si} has lower priority than τ_{tj}, (b) τ_{si} does not have a non-preemption relation to τ_{tj}, and either (c1) τ_{si} and τ_{tj} are in different transactions (and can possibly intersect due to unknown phasing), (c2) τ_{tj} can be delayed by jitter, possibly starting after the next invocation of τ_{si}, or (c3) τ_{si} can possibly finish after the start of the next invocation of τ_{tj}. Formally, the relation can be defined as follows:

$$\begin{aligned} \tau_{si} \prec \tau_{tj} \equiv \overbrace{P_{si} < P_{tj}}^{\text{(a)}} \wedge \overbrace{\neg\text{NOPRE}(\tau_{si}, \tau_{tj})}^{\text{(b)}} \\ \wedge \quad \big(\underbrace{s \neq t}_{\text{(c1)}} \vee \underbrace{J_{sj} > \psi_{sji}}_{\text{(c2)}} \vee \underbrace{R_{si} - O_{si} > T_s - \psi_{sji}}_{\text{(c3)}} \big) \end{aligned} \tag{8}$$

Lemma 2 *The $\prec$ relation is a safe approximation of the possible preemptions between tasks. That is, if τ_{si} can under any run-time circumstance be preempted by τ_{tj}, then $\tau_{si} \prec \tau_{tj}$ holds.*

Proof of Lemma 2 *If an instance υ_k of τ_{si} is preempted by an instance υ_l of τ_{tj}, then we must have $P_{si} < P_{tj}$, $\neg\text{NOPRE}(\tau_{si}, \tau_{tj})$ and $st_k < st_l < ft_k$. From the assumption about the resource protocol, we know that τ_{si} can not start between rt_l and st_l, and thus we must have $st_k < rt_l$.*

If $s \neq t$, then $\tau_{si} \prec \tau_{tj}$ holds. Thus, for the remaining proof we assume $s = t$, and consider two cases:

***Case 1:** If $O_{si} < O_{sj}$, then $\psi_{sji} = T_s + O_{si} - O_{sj}$.*
If $\alpha_k \leq \alpha_l$, we have
$st_l < ft_k \Rightarrow \alpha_l + O_{sj} < \alpha_k + R_{si} \Rightarrow$
$O_{sj} - O_{si} < R_{si} - O_{si} \Rightarrow T_s - \psi_{sji} < R_{si} - O_{si}$.
If $\alpha_k > \alpha_l$, then $\alpha_l + T_s \leq \alpha_k$, and we have
$st_k < rt_l \Rightarrow \alpha_k + O_{si} < \alpha_l + O_{sj} + J_{sj} \Rightarrow$
$\alpha_l + T_s + O_{si} < \alpha_l + O_{sj} + J_{sj} \Rightarrow$
$T_s + O_{si} - O_{sj} < J_{sj} \Rightarrow \psi_{sji} < J_{sj}$.

***Case 2:** If $O_{si} \geq O_{sj}$, then $\psi_{sji} = O_{si} - O_{sj}$.*
If $\alpha_k \geq \alpha_l$, we have
$st_k < rt_l \Rightarrow \alpha_k + O_{si} < \alpha_l + O_{sj} + J_{sj} \Rightarrow$
$\alpha_l + O_{si} < \alpha_l + O_{sj} + J_{sj} \Rightarrow$
$O_{si} - O_{sj} < J_{sj} \Rightarrow \psi_{sji} < J_{sj}$.
If $\alpha_k < \alpha_l$, then $\alpha_k + T_s \leq \alpha_l$, and we have
$st_l < ft_k \Rightarrow \alpha_l + O_{sj} < \alpha_k + R_{si} \Rightarrow$
$\alpha_k + T_s + O_{sj} < \alpha_k + R_{si} \Rightarrow$
$T_s + O_{sj} - O_{si} < R_{si} - O_{si} \Rightarrow$
$T_s - \psi_{sji} < R_{si} - O_{si}$.
In all four subcases, we either have $T_s - \psi_{sji} < R_{si} - O_{si}$ or $\psi_{sji} < J_{sj}$, which means that $\tau_{si} \prec \tau_{tj}$ holds. □

4.2. Possible preemption chains

A sequence Q of tasks is a *possible preemption chain* (PPC) if it holds that $\tau_{si} \prec \tau_{tj}$ for all τ_{si}, τ_{tj} in Q where τ_{si} 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., $SU_Q = \sum_{\tau_{si} \in Q} S_{si}$.

Lemma 3 *If $PC = \{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ is a preemption chain, and $Q = \{\tau_{s_1 i_1}, \tau_{s_2 i_2}, \ldots, \tau_{s_k i_k}\}$ is a corresponding sequence of tasks such that $\upsilon_q \in PC$ is an instance of $\tau_{s_q i_q}$, then Q is a PPC.*

Proof of Lemma 3 *For all task instances υ_p, υ_q in a preemption chain PC, if $p < q$ then it holds that $st_p < st_q < ft_p$. Since υ_p and υ_q are instances of $\tau_{s_p i_p}$ and $\tau_{s_q i_q}$ respectively, Lemma 2 implies that $\tau_{s_p i_p} \prec \tau_{s_q i_q}$, and thus Q is a PPC.* □

A PPC Q for which no other PPC have a higher stack usage in the same system is called a *maximal stack usage* PPC, or more informally, a *maximal* PPC. The stack upper bound problem can now be stated as finding a maximum stack usage PPC. We refer to this as the MAXPPC problem. We now prove that the stack usage of a maximal PPC Q in a system Γ is a safe upper bound on the stack usage of Γ.

Theorem 1 *The stack usage of a maximal PPC Q is a safe upper bound on the actual worst case stack usage for a system Γ.*

Proof of Theorem 1 *Let Ψ be the sequence of tasks instances participating in the preemption situation which cause the worst case stack usage, that is, $SWC = \sum_{\tau_{si} \in \Psi} S_{si}$. According to Lemma 3, we have that Ψ is a PPC with $SU_\Psi = SWC$. Since Q is a maximal PPC, $SU_\Psi \leq SU_Q$, which concludes the proof.* □

5. Algorithms

In [14], we proposed a polynomial method for stack analysis of a single transaction of the type described in Section 3. The polynomial time behavior of this method comes from the fact that a single transaction represented by tasks with offsets and response times can be efficiently analyzed using specialized graph algorithms [16]. These algorithms cannot be directly applied to analysis of a global stack shared by several transactions. When analyzing a single transaction in isolation, the task offsets and response times can be used to bound the time interval within which the tasks will execute. However, when several transactions are considered, we no longer have a common activation time, and therefore the graph algorithms used in [14] are no longer applicable. We therefore propose to analyze smaller

systems using an exact algorithm, guaranteed to find a maximal PPC. For larger systems, we propose to use a polynomial approximation, described in Section 5.4.

5.1. An exact algorithm for the MAXPPC problem

We solve the problem of finding a maximal PPC by forming a (directed) *preemption graph* of nodes representing tasks, and edges representing possible preemptions, as defined in (8). An example taskset (assuming $J = B = 0$ and $\neg\text{NOPRE}(\tau_{si}, \tau_{tj})$ for all tasks) and the corresponding preemption graph is shown in Figure 2.

Task	O	P	R	S
τ_{11}	0	1	3854	4
τ_{12}	1697	2	3837	1
τ_{13}	4635	4	4781	2
τ_{21}	0	3	393	2
τ_{22}	617	1	3854	1
τ_{23}	2588	3	3699	3

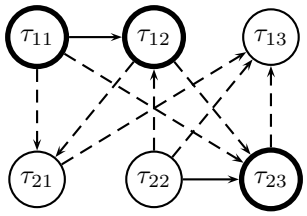

Figure 2. An example of a preemption graph, where solid edges represent possible preemptions within a transaction, and dashed edges represent possible preemptions between different transactions. The tasks in a maximal possible preemption chain are marked.

The method is based on a branch-and-bound search for PPCs in this graph, recursively building PPCs Q^i. An outline of the algorithm is given in Algorithm 1. Initially, $Q^0 = M = \emptyset$. If in any recursive step the total stack usage SU_{Q^i} is greater than the stack usage SU_M of the current maximal PPC M, then Q^i becomes the new maximum. We define a *cover set* $C(Q)$ of a PPC Q as a set of tasks for which all tasks in $C(Q)$ can possibly preempt all tasks in Q. A cover set is *maximal* if it cannot be extended by any other task. The algorithm maintains $C(Q^i)$ and then recursively examines an extension $Q^{i+1} = Q^i \cup \{\tau_{tj}\}$ of Q^i for each task τ_{tj} in $C(Q^i)$. We also apply a bounding function UB to terminate search in branches which clearly cannot contain a maximum PPC. This bounding function is further discussed in Section 5.2.

Algorithm 1: Computing a maximal PPC in a generic preemption graph.

MAXPPC(Q)
(1) **if** $SU_Q > SU_M$ **then** $M \leftarrow Q$
(2) $C(Q) = \{\tau_{tj} \mid \forall \tau_{si} \in Q . \tau_{si} \prec \tau_{tj}\}$
(3) **if** $SU_Q + \text{UB}(C(Q)) \leq SU_M$ **then return**
(4) **foreach** $\tau_{tj} \in C(Q)$
(5) MAXPPC($Q \cup \{\tau_{tj}\}$)

In Section 5.3, we show that the *algorithm* described is *exact* in the sense that it always computes the maximal PPC, and therefore solves the MAXPPC problem. We also show that the method is *safe*, because the stack usage of the maximal PPC is an upper bound on the stack usage of the system.

Note that our method of *stack bounding* is not exact, since the $\prec$ relation is in itself a (safe) approximation. Also, there are other factors unaccounted for. For example, there may be further restrictions on the possible nesting patterns due to mutual exclusion, and the tasks may not use their maximum stack when interrupted.

5.2. Bounding the maximal PPC

Choosing the right function for the bounding step in Algorithm 1 is essential to guarantee correctness and to acquire a method usable in practice. A *safe upper bound function* on the maximal PPC stack usage for a set of tasks C is a function UB for which $\text{UB}(C) \geq SU_K$, where $K \subseteq C$ is a maximal PPC. We use the most stack-intensive path in the preemption graph spanned by $C(Q)$ as the UB function, which we refer to as the PUB method. A heaviest path (w.r.t. stack space) in a directed acyclic graph can be found in $O(n+m)$ time, where n is the number of vertices and m is the number of edges [7].

Theorem 2 *PUB is a safe upper bound function on the maximal PPC stack usage.*

Proof of Theorem 2 *From the definition of a PPC in Section 4.2, we have that a maximal PPC $K \subseteq C$ is a path with stack usage SU_K. PUB results in the maximum stack usage of any path $A \subseteq C$. Therefore, $PUB(C) \geq SU_K$, which concludes the proof.* □

We refer to the combination of the branch-and-bound method in Algorithm 1 with the most stack-intensive path relaxation (PUB) as bounding function, as the PPCBB algorithm.

5.3. Correctness

In order to claim correctness of Algorithm 1 we need to show that it computes a maximal PPC. Theorem 1 then gives us that the stack usage of the PPC computed by Algorithm 1 is an upper bound on the stack usage of the system. We first need to prove a lemma regarding the stack usage of a PPC when extended with tasks from a cover set.

Lemma 4 *If Q is a PPC, $C(Q)$ is a cover set of Q, and $K \subseteq C(Q)$ is another PPC, then $Q \cup K$ is a PPC with stack usage $SU_{Q \cup K} = SU_Q + SU_K$.*

Proof of Lemma 4 *All tasks in Q can be preempted by all tasks in $C(Q)$, and since Q and K are both PPCs and $K \subseteq C(Q)$, $Q \cup K$ is a PPC. Furthermore, $Q \cap C(Q) = \emptyset$ because no task can preempt itself, and thus $Q \cap K = \emptyset$, and $SU_{Q\cup K} = \sum_{\tau_{si} \in Q \cup K} S_{si} = \sum_{\tau_{si} \in Q} S_{si} + \sum_{\tau_{tj} \in K} S_{tj} = SU_Q + SU_K$.* □

We can now prove that Algorithm 1 is correct.

Theorem 3 *If* UB *is a safe stack usage upper bound function, then Algorithm 1 terminates with a maximal PPC.*

Proof of Theorem 3 *The proof is given in two parts.*

We first assume that $\text{UB}(C) = \infty$ *for all inputs C, so that Algorithm 1 never returns on line (3). Given a PPC Q and any task τ_{tj} from a maximal cover set $C(Q)$, we can form a new set $Q' = Q \cup \{\tau_{tj}\}$ which is also a PPC (from Lemma 4). Therefore, Q is always a PPC, and since the algorithm extends Q with one task from $C(Q)$ and $Q \cap C(Q) = \emptyset$, the algorithm will terminate. If τ_{si} is not in $C(Q)$, then $Q \cup \{\tau_{si}\}$ is not a PPC. All together, the algorithm explores all PPCs, including a maximal PPC which will be stored in M and consequently returned when the algorithm terminates.*

Now assume that UB *is a safe upper bound function on the maximal PPC stack usage in a set of tasks. From Lemma 4 we have $SU_{Q\cup K} = SU_Q + SU_K$ for all PPCs $K \subseteq C(Q)$, and subsequently this also holds if K is a maximal PPC in $C(Q)$, in which case $Q \cup K$ is also a maximal PPC in $Q \cup C(Q)$ (from the definition of cover set). Since* UB *is safe, $SU_Q + \text{UB}(C(Q)) \geq SU_{Q\cup K}$. Thus, if $SU_Q + \text{UB}(C(Q)) \leq SU_M$ where M is the most stack-intensive PPC found so far, there is no PPC in $Q \cup C(Q)$ which has a higher stack usage than M, and we can return from this branch without losing any maximal solutions.* □

5.4. Polynomial approximations

Algorithm 1 is non-polynomial. In Section 6, we show that despite this, the algorithm can be used to analyze realistically sized task-sets. However, an exponential analysis method will still be too time-consuming for practical use when the number of tasks under analysis is too large. We therefore propose a polynomial time approximation for analyzing stack size where the number of tasks is too high to be analyzed using the branch-and-bound method. The approximation is a combination of two methods. The first one, STLA, is based on analysis of individual transactions in isolation, and essentially uses the sum for all transactions as an upper bound on the total stack consumption. The method is described in [14], but has been modified for the current task model, to account for precedence constraints and to allow response times larger than the period. STLA is a safe upper bound if the analysis for each transaction is safe, and runs in $O(kn^3)$ time, where k is the number of transactions, and n is the maximal number of tasks in a single transaction.

STLA is overly pessimistic in situations where equally prioritized stack-intensive tasks in different transactions influence the isolated transaction stack upper bound. Since the tasks have equal priority, they cannot preempt each other, and the global upper bound obtained is pessimistic. To remedy this, we also use a second polynomial method to obtain a different upper bound. The method, called PUB, finds a maximum stack usage path in the global preemption graph of all tasks in the system, and is the same one described in Section 5.2 and used as an upper bound function in PPCBB.

To illustrate the difference between PPCBB, STLA and PUB, consider the task set illustrated in Figure 2. The maximal PPC in this task set is $\{\tau_{11}, \tau_{12}, \tau_{23}\}$ with a total stack usage of 8. This is the result that PPCBB would return. In contrast, STLA would compute an upper bound by considering the two transactions $\Gamma_1 = \{\tau_{11}, \tau_{12}, \tau_{13}\}$ and $\Gamma_2 = \{\tau_{21}, \tau_{22}, \tau_{23}\}$ in isolation, computing the PPCs $\{\tau_{11}, \tau_{12}\}$ with stack usage 5 for Γ_1 and $\{\tau_{22}, \tau_{23}\}$ for Γ_2 with a stack usage of 4. The sum, 9, would be returned as the result. Finally, PUB would return the stack usage 10 of the most stack intensive path $\{\tau_{11}, \tau_{12}, \tau_{23}, \tau_{13}\}$ in the graph, which is not a PPC but is nonetheless, as shown in the proofs of Theorem 1 and 2, a safe approximation on the stack usage of the system.

Since $\tau_{si} \prec \tau_{tj} \rightarrow P_{si} < P_{tj}$, a stack usage path P can never include two tasks on the same priority level. If we would relax the $\prec$ relation into $\prec^2 \equiv P_{si} < P_{tj}$, the stack usage of the most stack intensive path would be equal to the maximum stack for each priority level in the system. Therefore, PUB is always at least as good as the traditional approach (SPL). We propose to use the minimum of PUB and STLA (referred to as STLA_PUB) as a polynomial time alternative to PPCBB. Since both PUB and STLA are safe, STLA_PUB is also safe.

6. Evaluation

We evaluate the efficiency of our proposed methods by generating random task sets and calculating the stack upper bounds. All tasks in each generated task set share one common stack. We use three methods (SPL, STLA_PUB, PPCBB) to calculate an upper bound on the shared system stack. Thus, the upper bounds are illustrated by the following graphs:

SPL: The traditional approach to determine an upper bound (sum of maximum stack usage of each priority/preemption level).

STLA_PUB: This represents the minimum of the polynomial methods STLA (analysis of individual trans-

action) and PUB (longest path in global preemption graph). See Section 5.4 for details.

PPCBB: Non polynomial branch-and-bound based method with longest path relaxation. See Section 5.1 and 5.2 for details.

6.1. Simulation setup

We run the stack analysis application on an Intel Pentium 4, 2.18 GHz with 512 MB of RAM. We generate random task sets as input to the stack analysis application. The task generator takes the following input parameters (default values represent the base configuration of each analysis):

Parameter	**Default**
Number of transactions	5
Number of tasks	60
Total system load	40%
Task priority (min–max)	1–32
Task stack usage (min–max)	128–2048 bytes
Probability of precedence	10%

Using these parameters, task sets with the following characteristics are generated:

- The period time T_s of each transaction is set to 10000.
- Each task offset (O_{si}) is randomly and uniformly distributed between 0 and $T_s/2$.
- Task priorities and the stack usages are assigned randomly between minimum and maximum value with a uniform distribution.
- The total system load, and the number of tasks in the system, is distributed among the transactions in such way that all transactions have the same amount of load and the same number of tasks.
- Worst case execution times, C_{si}, are initially assigned to each task in such way that tasks are separated in time within a transaction. The execution times are then adjusted by a fraction, so that the the total system load (as defined by the input parameter) is obtained, preserving time separation of tasks within a transaction.
- Each task is assigned a precedence relation with a probability specified by the precedence probability attribute. For example, if the probability of precedence for τ_{si} is 10%, then for each succeeding task (i.e., task with larger or equal offset than τ_{si}) in Γ_s, there is a 10% probability that τ_{si} is given precedence over the task. When all precedences are assigned, transitive precedences are established, e.g, if τ_{si} has precedence over τ_{sj} and τ_{sj} has precedence over τ_{sk}, then τ_{si} has precedence over τ_{sk}.
- We assign deadline $D_{si} = T_s$ for each task. All tasks are required to meet their deadlines (otherwise the task set is considered unschedulable). In case the generated task set is unschedulable, the task set is discarded and a new task set is generated.

6.2. Results

Each point in the graphs represents the mean stack usage of 100 randomly generated schedulable task sets. For each point, a confidence interval (confidence level of 95%) is shown to indicate the reliability of the figures. For each diagram, we vary one parameter, keeping all other parameters according to the base configuration. In addition to calculating upper bounds, we also measured the mean execution time for each method. The mean execution times for SPL lies in the range of micro seconds, for STLA_PUB the mean execution time lies in the in the range of milliseconds and for PPCBB in the range from milliseconds up to five seconds.

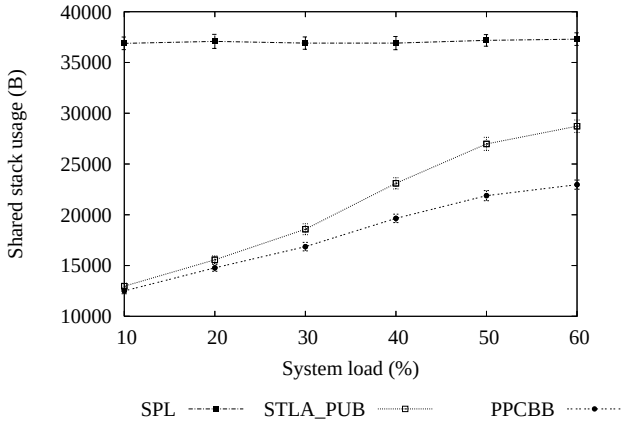

Figure 3: Varying system load

In Figure 3 we vary the total system load from 10% to 60%. As expected, the stack upper bound using the traditional method (SPL) is constant and unaffected by variations in load. This is due to the fact that SPL only considers priorities when calculating the upper bound. Both STLA_PUB and PPCBB produces upper bounds that are slowly increasing with increasing load. This is natural, since increasing the load, keeping all other parameters according to the base configuration, typically results in larger response times, which in turn increases the number of possible preemptions in the system.

In Figure 4 we vary the maximum priority of tasks from 1 to 64. This gives a possible priority distribution of 1 to *n*, where *n* is indicated by the x-axis. We observe that for small values on *n* the difference between the methods is small. For larger values on *n* the difference is significant.

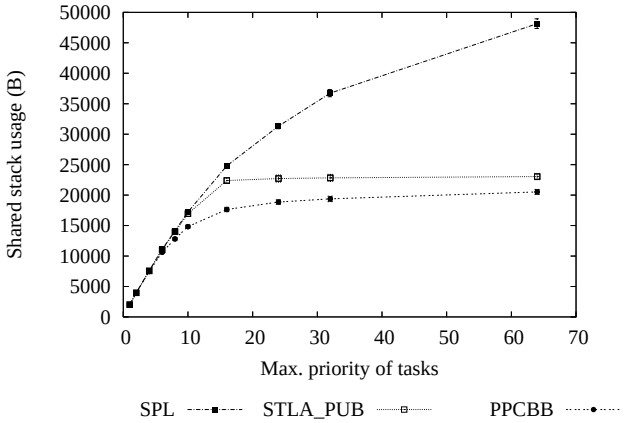

Figure 4: Varying maximum priority

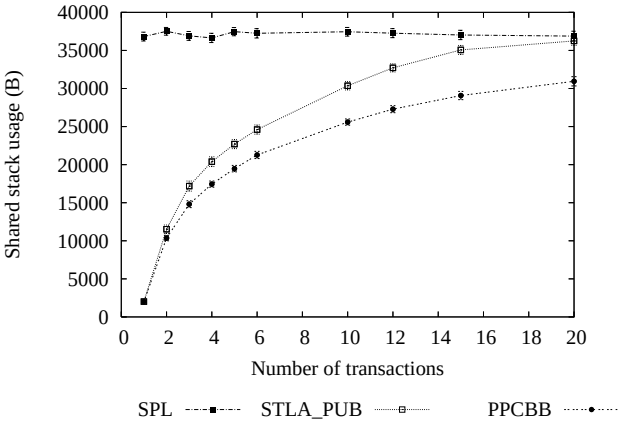

Figure 6: Varying the number of transactions

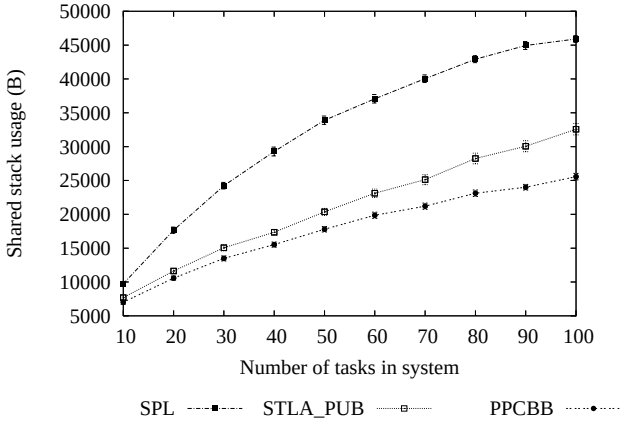

Figure 5: Varying the number of tasks in the system

In Figure 5 we vary the number of tasks in the system from 10 to 100. With a low number of tasks in the system, there is a larger possibility that tasks have unique priorities hence considered to be part of a preemption chain by SPL. SPLA_PUB and PPCBB goes one step further and examines preemption patterns, with a tighter upper bound as a result.

In Figure 6 we vary the number of transactions from 1 to 20. We see that both SPLA_PUB and PPCBB increase when increasing the number of transactions in the system. With more transactions, the arbitrary phasing between them increases the possibility of nested preemptions, resulting in increased shared stack usage. SPL is constant and unaffected by variation in the number of transactions. Again, this is expected, since SPL only considers priorities.

7. Conclusions and future work

Allowing tasks to share a common run-time stack can reduce the amount of RAM needed for an application. However, in order to safely reduce the overall run-time stack, one must be able to analyze possible preemption patterns statically. And from those preemtion patters deduce the possible stack-usage. Static information such as the system model, scheduling policy, and run-time mechanism can be used to constrain the values of the dynamic task-properties that affect shared stack usage.

A task model with such static information is the task model with offsets (the transactional task model) where priorities, offsets and precedences limit the possible preemption patterns. We have, for that task model, developed a system wide preemption analysis that safely approximates the actual preemptions and forms a basis for safe upper bound of the total shared stack usage.

We presented two novel methods for determining a safe upper bound on the stack usage. Both methods analyze a graph consisting of tasks and possible preemptions between these. The first method is an exact search for maximal possible preemption chains. The second method is a combination of two algorithms, both being polynomial approximations of the first. We formally showed that both methods are safe in the sense that they will never underestimate the amount of stack space needed.

The methods have a clear practical value in a verification/validation phase of system development. They can be used to formally verify that stack overflow will not occur during run time. In a simulation study, we evaluated our techniques and compared it to the traditional method to estimate stack usage. We found that our exact method significantly reduced the amount of stack memory needed. In our simulations, a decrease in the order of 40% was typical, with a runtime in the order of seconds. Our polynomial approximation consequently yields about 20% higher bound than the exact method.

In future work our methods can be used to further reduce the stack bound by more detailed modeling of the sys-

tem behavior. For example, the assumption that each task uses its maximum stack when preempted may lead to overly pessimistic result if the stack usage is highly variable during execution. With knowledge about the variation of stack usage, one might split a task into several segments, each with its own stack usage. These segments can then be modeled as separate tasks with precedence constraints, and thus we should obtain a tighter bound on the stack usage. Furthermore, a similar technique could also be used to split up a task that uses shared resources where the part that uses the resource can be modeled as a task with a mutual exclusion relation to other tasks that uses the same resource.

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

[6] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. Henzinger, and J. Palsberg. Stack size analysis for interrupt-driven programs. In *Proceedings of the 10th Annual International Static Analysis Symposium*, June 2003.

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

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

[9] 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.

[10] Evidence Srl. Web page, http://www.evidence.eu.com.

[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] R. Ghattas and A. Dean. Preemption threshold scheduling:stack optimality, enhancements and analysis. In *Proceedings of the 13th IEEE REal-Time and Embedded Technology and Applications Symposium*, April 2007.

[13] 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.

[14] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Determining maximum stack usage in preemptive shared stack systems. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, Dec 2006.

[15] 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.

[16] T. A. McKee and F. McMorris. *Topics in intersection graph theory*. Monographs on Discrete Mathematics and Applications #2. SIAM, 1999.

[17] Micro Digital. Web page, http://www.smxinfo.com/mt.htm.

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

[19] 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.

[20] 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.

[21] 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.

[22] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems*, 4(4):751–778, Nov 2005.

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

[24] 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.

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

[26] 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.

[27] Unicoi Systems. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.