Refined SRP Stack Memory Analysis by Exploiting Critical Sections for Shared Resources

Johan Eriksson, Simon Aittamaa, Per Lindgren

▶ **To cite this version:**

Johan Eriksson, Simon Aittamaa, Per Lindgren. Refined SRP Stack Memory Analysis by Exploiting Critical Sections for Shared Resources. Embedded Real Time Software and Systems (ERTS2014), Feb 2014, Toulouse, France. ⟨hal-02272330⟩

HAL Id: hal-02272330

https://hal.science/hal-02272330v1

Submitted on 27 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Refined SRP Stack Memory Analysis by Exploiting Critical Sections for Shared Resources

Johan Eriksson Simon Aittamaa Per Lindgren
EISLAB, Luleå University of Technology, 97187 Luleå
Email: {Johan.Eriksson, Simon.Aittamaa, Per.Lindgren}@ltu.se

December 13, 2013

Abstract

Embedded (real-time) systems often operate on platforms with limited memory resources. In particular, the size of the random access memory (RAM) may be a limiting factor. Scheduling under the Stack Resource Policy offers (among other benefits) single stack execution, which reduces the memory requirement in cases where shared resources limits potential preemptions.

In this paper we refine previous results on stack memory analysis for SRP based systems. For a task J we associate the section of code in between a resource request and release of R to a sub-task J_R. If J_R is implemented as a function, the stack usage of J_R can be accounted for locally (and not part of the initial allocation for J as done in previously published work.)

The proposed method has been been implemented and evaluated on a set of synthetic benchmarks. Our experiments show a clear impact of per sub-task stack allocation to reducing stack memory requirement. The proposed stack memory analyzer makes no approximations, still run-time is shown to be in the range of 1 minute, for 50 tasks and 250 subtasks. At the same time the accuracy of our method is improved in comparison to previous work.

Co-funded by the European Union, European Regional Development Fund (ERDF)

1 Introduction

Embedded systems often operate under both timing and resource constraints. Tasks (activities) in such systems are typically concurrent, triggered by some event (emitted by the environment, e.g. a hardware timer, or raised internally). Preemptive scheduling may improve responsiveness for such systems [1]. Preemption amounts to storing the context of the preempted task and restoring (or creating) the context of the task to execute. A common abstraction is to associate each task to a thread, with a pre-allocated memory region for the execution stack [2]. The total stack space usage is simply calculated from the sum of the (maximum) stack usage for each thread. Context switches can be done in arbitrary order (which allows for full flexibility for scheduling).

However, by limiting/restricting the flexibility of the scheduler (and refining the analysis), the assessed (worst case) stack memory usage can be reduced.

An alternative approach for preemptive execution of task sets with shared resources is offered by the Stack Resource Policy (SRP) [3], under which tasks (or jobs) execute on a single stack. Hence, using SRP stack space for individual jobs do not need to be pre-allocated. The total amount of memory required will be less or equal to that of a thread based equivalent, and may in the presence of shared resources be significantly lower. SRP imposes the restriction that tasks/threads may never run preemptively w.r.t to each other if assigned the same priority, and that a task A will never preempt another task B that holds a resource that would force A to wait until B releases the

resource (i.e, a task, when executing, will be able to finish without having to wait for any other task).

Recent work [4] incorporate job offsets (temporal dependencies) in the analysis to improve the precision (taking only the feasible set of preemptions into account) when assessing the worst case stack depth. The work of [4] includes an approximative algorithm, speeding up calculations, such to allow larger task-sets to be analysed.

In this paper we further refine the results on offset analysis by considering stack usage on a per sub-task basis (extending the per task basis analysis of [4].) For a task J we associate the section of code in between a resource request and release of R to a sub-task J_R, which in turn may contain a hierarchy of sub-tasks. If J_R is implemented as a function, the stack usage of J_R can be accounted for locally. As an sub-task J_R, holding the resource R is only subject to a subset of possible pre-emptions for an outer task, the maximum required stack space will be reduced (in comparison to considering the stack space for a task J where all J_Rs are inlined).

The (potential) impact of the proposed method is evaluated on a set of synthetically generated test cases. The experiments show that significant reductions can be obtained by accounting for stack usage on per sub-task basis compared to per task basis previously presented.

Furthermore, on another set of experiments we showcase the exponentially growing execution time (measured in number of iterations) when varying the number of tasks. Despite the complexity, our experiments validate the feasibility the proposed method. Our experimental implementation uses about a minute run-time on a 3GHz core 2 duo (single threaded implementation) to calculate stack depth of 50 main tasks with a total number of 250 sub-tasks, which in practise suffices for many embedded applications.

In order to reduce the execution time, we deploy a simple cache scheme. Even with only two cached results for each task, the execution time (number of iterations) is reduced by a factor of more than 20. Notice, while caching reduces execution time in practise, the exponential complexity still remains. In contrast to [4] the proposed method does not introduce any internal approximations,

i.e. it gives the exact solution for the given input. However, for larger task-sets an exact approach becomes computationally intractable and approximative solutions may be developed for assessing stack depth on basis of sub-tasks and is a set topic for future work.

2 Formal System Model

As a basis, we use the formal system model with static offsets [5], where we have k transactions, each Γ_i activated with a minimum inter-arrival time T_i, and consists of $|\Gamma_i|$ number of tasks, where each task is define by a tuple where, C denotes the execution time, O the offset for release (relative to the release of the transaction), D its relative deadline, J maximum jitter, B maximum blocking from lower priority tasks, P priority, and S its stack usage.

$$\begin{aligned} \Gamma &:= \{\langle \Gamma_1, T_1 \rangle, \ldots, \langle \Gamma_k, T_k \rangle\} \\ \Gamma_i &:= \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\} \\ \tau_{ij} &:= \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij}, S_{ij} \rangle \end{aligned} \tag{1}$$

We assume the same restrictions as in [5] unless else stated (e.g., the task set is scheduleable).

In this model, the stack usage is defined on a per-task basis. We refine each task to consist of a main task and set of sub-tasks (associated with a local stack space and resource) in a hierarchical manner. This model goes hand in hand with the task model defined by Baker for the Stack Resource Policy [3]. We assume single unit resources, and define the preemption level equal to the priority. We statically compute the (current) resource ceilings as described in [6]. We define our tasks with sub-tasks as:

$$\tau_{ij[x]} := \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, PR_{ij[x]}, S_{ij[x]}, sub_{ij[x]} \rangle \tag{2}$$

$$sub_{ij[x]} := \{\tau_{ij[x,1]}, \ldots, \tau_{ij[x,m]}\} \tag{3}$$

E.g., the definitions

$$
\begin{aligned}
\tau_{a[]} &:= \langle C_a, O_a, D_a, J_a, B_a, PR_{a[]}, S_{a[]}, sub_{a[]}\rangle \\
sub_{a[]} &:= \{\tau_{a[1]}, \tau_{a[2]}\} \\
\tau_{a[1]} &:= \langle C_{a[1]}, O_{a[1]}, D_{a[1]}, J_a, B_a, PR_{a[1]}, S_{a[1]}, sub_{a[1]}\rangle \\
\tau_{a[2]} &:= \langle C_{a[2]}, O_{a[2]}, D_{a[1]}, J_a, B_a, PR_{a[2]}, S_{a[2]}, \{\}\rangle \\
sub_{a[1]} &:= \{\tau_{a[1,1]}, \tau_{a[1,2]}\} \\
\tau_{a[1,1]} &:= \langle C_{a[1,1]}, O_{a[1,1]}, D_{a[1,1]}, J_a, B_a, PR_{a[1,1]}, S_{a[1,1]}, \{\}\rangle \\
\tau_{a[1,2]} &:= \langle C_{a[1,2]}, O_{a[1,2]}, D_{a[1,2]}, J_a, B_a, PR_{a[1,2]}, S_{a[1,2]}, \{\}\rangle \\
S_{a[]} &:= 16 \\
PR_{a[]} &:= P_{a[]} = 1 \\
S_{a[1]} &:= 24 \\
PR_{a[1]} &:= \lceil r_1 \rceil = 2 \\
S_{a[2]} &:= 20 \\
PR_{a[2]} &:= \lceil r_2 \rceil = 3 \\
S_{a[1,1]} &:= 32 \\
PR_{a[1,1]} &:= \lceil r_3 \rceil = 4 \\
S_{a[1,2]} &:= 8 \\
PR_{[1,2]} &:= \lceil r_2 \rceil = 3
\end{aligned} \tag{4}
$$

In the example τ_a, has the base priority 1, and hence pre-emption level 1 ($PR_{a[]} = 1$). The task has two immediate sub-tasks $\{\tau_{a[1]}, \tau_{a[2]}\}$, where $\tau_{a[1]}$ holds the resource r_1 for the duration $C_{a[1]}$. As $\tau_{a[1]}$ is a sub-task, release jitter and blocking are inherited from the parent task J_a. Deadline and offset can be calculated using information from the list of sub-tasks and the parent task. An approximation is obtained by simply inheriting the parents execution window. A refinement of the deadline $D_{a[1]} = D_a - C_{a[2]}$, takes the execution time of subsequent sub-tasks into account, similarly a refinement of the offset $O_{a[2]} = O_a + C_{a[1]}$, takes the execution time of preceding sub-tasks into account. Further refinements may be possible but is not scope for the presented work.

The (sub) task $\tau_{a[1]}$ in turn consists of two sub-tasks $\{\tau_{a[1,1]}, \tau_{a[1,2]}\}$, holding the resources r_3 and r_2, giving pre-emptions levels $PR_{a[1,1]} = 4$, and $PR_{a[1,2]} = 3$ respectively. Deadlines and offsets are derived as described above.

3 Preemption criteria

Our method builds on the job admission criteria from Baker [3], and can be formulated as an extension of equation 6 in [4]. We define W_{ijx} as the scheduling window for a (sub) task τ_{ijx}, where $start(W_{ijx})$ and $end(W_{ijx})$ gives the bounding (integer) values relative to the release of Γ_i. Basic computations for $start$ and end are given in [7], but may be further improved on by utilising the refined deadlines and offsets as discussed above.

The preemption criteria is defined in (5).

$$
\begin{aligned}
&\tau_{tix} \prec \tau_{tiy} \equiv \tau_{tiy} \in sub_{tix} \\
&\tau_{tix} \prec \tau_{tj[]} \equiv \\
&\quad (start(W_{tix}) < start(W_{tj[]}) < end(W_{tix}) \wedge PR_{tix} < PR_{tj[]}) \\
&\tau_{tix} \prec \tau_{uj[]} \equiv PR_{tix} < PR_{uj[]}
\end{aligned} \tag{5}
$$

E.g., in Equation 4, $\tau_a[]$ may be preempted by the critical sections (sub-tasks) $\tau_a[1]$ and $\tau_a[2]$ (associated with r_1 and r_2 respectively), but not by the critical sections $\tau_a[1,1]$ and $\tau_a[1,2]$ (associated with r_3 and r_2 respectively) since they are not in the set $sub_{a[]}$ of immediate siblings to $\tau_a[]$. $\tau_a[1,1]$ and $\tau_a[1,2]$ cannot be executed unless $\tau_a[1]$ (the critical section r_1) is executing, hence they cannot preempt $\tau_a[]$ directly. This corresponds to the first criterion of Equation 5.

Assume we have another task $\tau_b[]$:

$$
\begin{aligned}
\tau_{b[]} &:= \langle C_b, O_b, D_b, J_b, B_b, PR_{b[]}, S_{b[]}, \{\}\rangle \\
S_{b[]} &:= 16 \\
PR_{a[]} &:= P_{b[]} = 4
\end{aligned} \tag{6}
$$

Assume that t_a and t_b, belongs to the same transaction and that $(start(W_{ax}) < start(W_{b[]}) < end(W_{ax})$ holds for $x = \{[], [1], [1,1]\}$, only $t_a[]$ and $t_a[1]$ may be preempted, since $(PR_{a[1,1]} = 4) \nless (PR_b = 4)$. This corresponds to the second criterion of Equation 5.

Assume that t_a and t_b, belongs to the different transactions, then $t_a[]$, $t_a[1]$, $t_a[1,2]$ and $t_a[2]$ may be preempted. This corresponds to the third criterion of Equation 5.

4 Implementation and Evaluation

The iterative implementation of the algorithm follows directly from the definition of the pre-emtion criteria, exhaustively searching for the worst case pre-emption pattern w.r.t., to accumulated stack depth.

To reduce computation time, we introduce a simple caching mechanism. The cache reduces the number of tests required by storing the last test done on a single task and if the same possible preemption paths are present the cached result is used. Further optimisations can be explored, e.g., deploying dynamic programming building on partial results, but it is beyond the scope of this paper. However, the time complexity is by the definition of the problem $O(2^n * 2^p)$, where n is the number of tasks (including sub-tasks) and p the number of preemptions. In practise p is bound by the number of hardware interrupt priority levels. The memory complexity is linear to the task-set and the number of cache entries per task (for the experiments we used the latest two results).

Our evaluation shows that it is feasible to use this exact test for realistically sized task sets. For example a test of 50 main tasks and 250 sub-tasks (critical sections) takes about 1 minute on a 3GHz core2-duo using unoptimised single threaded python code with only simple caching and pre-calculations.

4.1 Example

For the given example we showcase the different stack estimation algorithms. Using the system model described in Section 2 and the system defined by (4) and (3) as example system, we demonstrate a number of approaches to estimating the maximum stack usage.

4.1.1 Method 1, SPL

In [4] (eq (1)), an algorithm, called SPL, for estimating stack usage for an SRP system is defined. The strength of this algorithm is the computational complexity, which is linear, but its estimate is overly pessimistic.

The algorithm works by identifying the task with maximum stack usage for each priority level and then sum all identified tasks.

The input to the algorithm is the priority level and maximum stack usage of all tasks in the system:

$$\tau_i := \langle PR_i, S_i \rangle \tag{7}$$

$$\begin{aligned} \tau_a &:= \langle 1, 72 \rangle \\ \tau_b &:= \langle 4, 16 \rangle \end{aligned} \tag{8}$$

The input, extracted from the example system, to the algorithm is shown in (8) and the result is $72 + 16 = 88$.

4.1.2 Method 2, exhaustive search

This algorithm extracts the lower bound for the stack usage if resource information is omitted. It is a recursive algorithm which exhaustively searches all valid preemption paths and returns the maximum (accumulated) stack-depth of all paths.

The input to the algorithm is all tasks in the system:

$$\tau_i := \langle PP_i, S_i \rangle \tag{9}$$

PP_i is the set of task that might preempt i, it is computed using Equation 6 in [4]. For the given example system the input is:

$$\begin{aligned} \tau_a &:= \langle \{\tau_b\}, 72 \rangle \\ \tau_b &:= \langle \{0\}, 16 \rangle \end{aligned} \tag{10}$$

The exhaustive search search shows that the most expensive path to be: $\tau_a \prec \tau_b$ With the accumulated stack depth of $72 + 16 = 88$.

4.1.3 Method 3, SPL with resource information

This method extends SPL to include subtask information. The input to the algorithm is the priority level and maximum stack usage of all tasks and sub-tasks in the system:

$$\tau_{i[x]} := \langle PR_{i[x]}, S_{i[x]} \rangle \tag{11}$$

For the given example system the input is:

$$
\begin{aligned}
\tau_a &:= \langle 1, 16\rangle \\
\tau_{a[1]} &:= \langle 2, 24\rangle \\
\tau_{a[2]} &:= \langle 3, 20\rangle \\
\tau_{a[1,1]} &:= \langle 4, 32\rangle \\
\tau_{a[1,2]} &:= \langle 3, 8\rangle \\
\tau_b &:= \langle 4, 16\rangle
\end{aligned} \tag{12}
$$

The identified tasks (the task with maximum stack usage for each preemption level) are: $\tau_a, \tau_a[1], \tau_a[2], \tau_a[1,1]$ stack usage can be calculated as $16 + 24 + 20 + 32 = 92$. Note, that in this case the result is worse than method 1, however in average this is not the case (see chapter 5).

4.1.4 Method 4, exhaustive search with resource information

Here we extend method 2 to utilize subtasks, i.e the input to the algorithm is computed as:

$$\tau_{i[x]} := \langle PP_{i[x]}, S_{i[x]}, sub_{ij[x]}\rangle \tag{13}$$

$PP_{i[x]}$ is the set of task that might preempt $i[x]$, it is computed using eq (5). For the given system model the input computes to: Applying (13) to the example system yields (14).

$$
\begin{aligned}
\tau_a &:= \langle \{\tau_b\}, 16, \{\tau_{a[1]}, \tau_{a[2]}\}\rangle \\
\tau_{a[1]} &:= \langle \{\tau_b\}, 24, \{\tau_{a[1,1]}, \tau_{a[1,2]}\}\rangle \\
\tau_{a[2]} &:= \langle \{\tau_b\}, 20, \{\}\rangle \\
\tau_{a[1,1]} &:= \langle \{\}, 32, \{\}\rangle \\
\tau_{a[1,2]} &:= \langle \{\tau_b\}, 8, \{\}\rangle \\
\tau_b &:= \langle \{\}, 16, \{\}\rangle
\end{aligned} \tag{14}
$$

The exhaustive search of the extended model yields that the worst-case stack usage is given by the path: $\tau_a \prec \tau_{a[1]} \prec \tau_{a[1,1]}$ With the accumulated stack depth of $16 + 24 + 32 = 72$. Method 4 is safe and tight, (it gives the true upper bound of the memory usage).

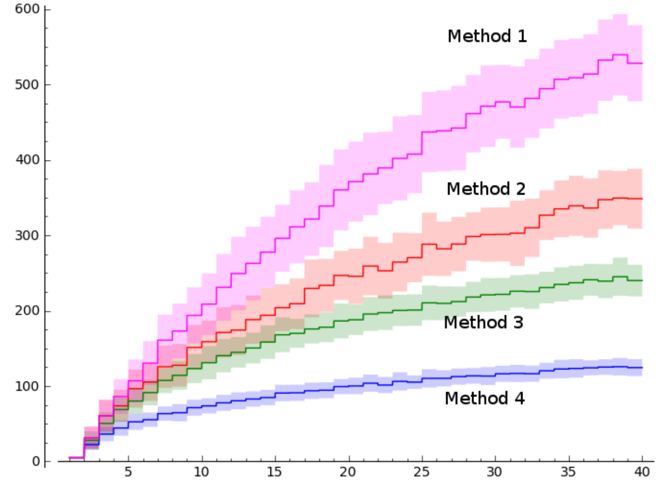

Figure 1: Calculated stack memory usage with the different algorithms. (X: Num of tasks, Y: Usage)

5 Synthetic Benchmarks

In order to evaluate the proposed approach to stack memory analysis, we synthesise a set of benchmarks. We generate 100 test cases per experiment to get statistical material for drawing initial conclusions. For the experiments, we have fixed the number of unique priority levels to 16. The number of tasks are spanned in between 1 and 40. Each task has a base priority between 1 and 16 (evenly distributed), between 1 and 10 critical sections (evenly distributed), where each critical section claims a resource with ceiling 0 to 3 higher than the base priority (saturating to 16). Each section of code allocates 5 stack units. The number of transactions in each test case is $log_2(n)$, (n number of tasks) with the distribution: 50% of tasks in transaction 1, 25% in 2, 12.5% in 3 etc. (until all tasks are assigned). The probability of overlapping in between tasks belonging to the same transaction is 70% (if not else stated).

Execution time is indicated in terms of number of iterations. Average results are shown as firm lines, while shaded areas denote the first standard deviation.

5.1 Experiment 1, Potential Stack Reduction

In a first set of experiments we showcase granularity of analysis, Figure 1. Method 1: The uppermost (pink) shows the SPL behaviour (summing up the maximum stack usage in each priority level) [4], which clearly shows the worst estimate (over approximation). Method 2: The next (counting from top/worst case) marked red, show the case where we assume that functions are inlined, i.e., stack space for sub-tasks are claimed as part of the parenting task. This is an exact method, however the input model is an over approximation since we assume that all stack (including sub-tasks) is allocated at the lowest preemption level (the base priority of the task). Method 3: The next (marked green), show the case where we consider sub-tasks as being functions, i.e., local allocation for each sub-task, and the maximum stack usage per preemtion level is accumulated. However, this is still an approximation, while we do not consider non-preemption information in-between tasks, and we do not consider relations between tasks and subtasks. Method 4: Bottom-most, (blue in figure), we show exact solution utilising non-preemption information and local stack allocation. The trend is clear, the exact solution for the task set is reducing the estimated stack space by a factor of 6 in comparison to SPL, and a factor of 3 to the per-task approach (used as a basis for the approximation work in [4]). Utilzing the non-preemption information and relations between subtasks and tasks improves the precision by more than a factor of two (blue compared to green).

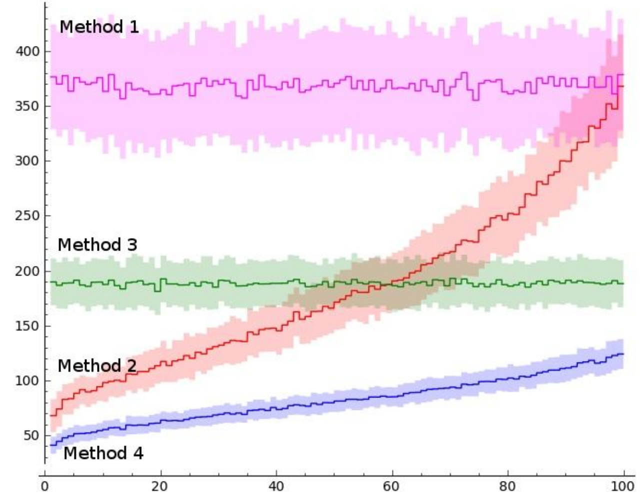

Figure 2: Varying probability of overlapping of tasks in a transaction. (X: Probability Y: Stack memory) Top to bottom: Method 1[

5.2 Experiment 2, Effect of non-preemptions

In the second set of experiments we show the effect of non-preemptions. We vary the probability of overlapping timing windows in between tasks in the same transaction from 0% to a 100% (Figure 2). Method 1: (purple) and Method 3: (green) do not utilise non-preemption information. Method 2: (red) relies only on the non-preemption information. Method 4: (blue), utilises both non-preemption and local (per sub-task) stack.

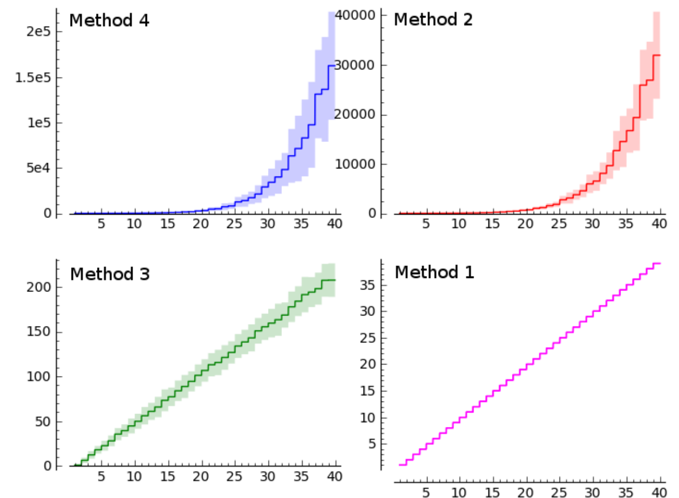

Figure 3: Uncached execution time, the number of tasks is varied from 1 to 40. 70 % chance of overlapping between tasks within a transaction.

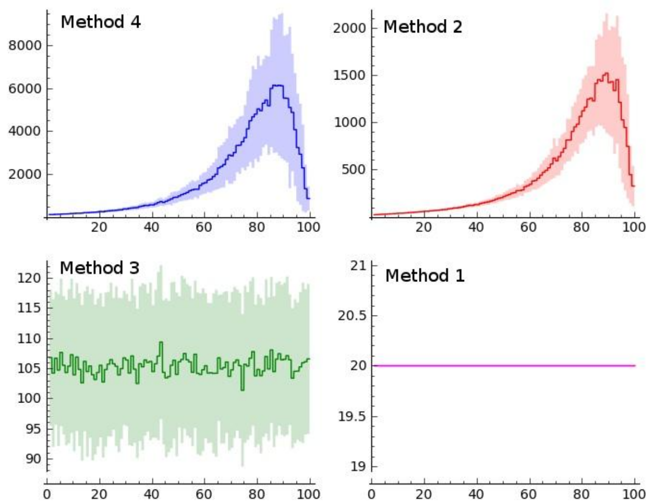

Figure 4: Cached execution time. The number of tasks is set to 40, and the probability of overlapping between tasks within a transaction is varied between 0% and 100 %.

5.3 Experiment 3, Uncached Execution Time

The third set of experiments (Figure 3) we showcase the execution time of the uncached implementation. Methods 2: (red) and 4: (blue) show exponential growth in number of iterations, while the Methods 1: and 3: are linear.

5.4 Experiment 4, Cached Execution time

The final set of experiments (Figure 4) we show the effect of simple caching of two latest results (with varying non-preemption set) computed for each task. In the experiment we use 40 tasks and vary the probability of overlapping timing windows. We observe a reduction by more than 20 times. The reason the number of iterations are reduced when closing 100% overlapping, is that the non-overlapping set shrinks towards the empty set, and hence the probability of cache hits increases. We expect more sophisticates caching and dynamic programming techniques to further reduce the execution time.

6 Conclusions and Future Work

We conclude that per-sub task stack analysis has significant impact to stack usage analysis, and that exact analysis is possible for task sets of practical use (50 tasks and 250 sub-tasks in a minute on an ordinary desktop). Further improvements can be made both in the direction of obtaining tighter results, and at the same time reducing execution time. This is possible by refining the pre-calculation of non-preemption sets (non-overlapping scheduling windows). (In the presented experimental implementation, we simply inherit the scheduling window.) While the solver in itself is exact the pre-calculation of scheduling windows is an over-estimation. The refinement will further reduce the execution time, since the non-preemption set will be extended (such preclude calculation of non-feasible preemptions). Future work also includes elaborating the trade-off in between in-lining of sub-tasks and sub-tasks as functions. The former reduces (constant) overhead of both CPU and stack for storing and restoring the (altered) set of registers (activation record), while the latter reduces the (dynamic) accumulated stack growth for sub-tasks.

References

[1] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," 1973.

[2] R. Barry, *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[3] T. Baker, "A stack-based resource allocation policy for realtime processes," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec. 1990, pp. 191 –200.

[4] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin, "Analysing stack usage in preemptive shared stack systems," Mälardalen University, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, July 2006. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1136

[5] 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, Tech. Rep., 1992.

[6] J. Eriksson, F. Haggstrom, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-Time For the Masses, Step 1: Programming API and Static Priority SRP Kernel Primitives," in *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013*, Jun. 2013.

[7] J. Eriksson, S. Aittamaa, J. Wiklander, P. Pietrzak, and P. Lindgren, "Srp-dm scheduling of component-based embedded real-time software," *First International Workshop on Dependable and Secure Industrial and Embedded Systems*, 2011.