Choosing Preemption Points to Minimize Typical Running Times

Sanjoy Baruah
Washington University in St. Louis

Nathan Fisher
Wayne State University

ABSTRACT

The problem of selecting "effective preemption points" in a program — points in the code at which to permit preemption — in order to minimize overall running time is considered. Prior solutions that have been proposed for this problem are based on workload models in which worst-case known upper bounds are assumed for the duration needed to perform preemptions at particular points in the code, and of the time needed to non-preemptively execute the code between preemption points. Since these solutions are based on worst-case assumptions, they tend to select effective preemption points in a conservative manner; consequently the overall execution time of the program may be needlessly large under most typical run-time circumstances. We consider a more general workload model in which "typical" values, as well as upper bounds, are assumed to be known for the preemption durations and the non-preemptive code-execution durations; given such information, we derive algorithms for the optimal placement of preemption points in a manner that minimizes the typical overall running time (while continuing to guarantee, if needed, upper bounds on the worst-case over-all running time). Both off-line solutions (in which all preemption points are selected prior to run-time) and on-line solutions (where the selection of some of the preemption points is made during run-time and therefore can exploit knowledge of the actual durations of prior preemptions and of the executions of already executed pieces of code) are presented and proved optimal.

CCS CONCEPTS

• **Computer systems organization → Real-time system specification**.

KEYWORDS

preemption placement, cache-related preemption delay, limited preemption, dynamic programming, online algorithms

ACM Reference Format:
Sanjoy Baruah and Nathan Fisher. 2019. Choosing Preemption Points to Minimize Typical Running Times. In *27th International Conference on Real-Time Networks and Systems (RTNS 2019), November 6–8, 2019, Toulouse, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3356401.3356407

RTNS 2019, November 6–8, 2019, Toulouse, France

ACM ISBN 978-1-4503-7223-7/19/11…$15.00
https://doi.org/10.1145/3356401.3356407

1 INTRODUCTION AND MOTIVATION

Many safety-critical systems are required to have their correctness validated prior to deployment. For real-time systems, the correctness criteria include guarantees that the system will meet specified deadlines during run-time. The likelihood of meeting all required deadlines in a system is generally enhanced if *preemptions* are permitted: i.e., a currently executing process may be interrupted in order to enable the execution of some other process (with, e.g., an earlier deadline). However, preemptions may incur considerable run-time overhead (for instance, preemptions may require that cache lines evicted by a preempting process be re-loaded, instruction pipelines invalidated, etc.); additionally, permitting preemptions restricts the usage of non-preemptable serially reusable resources to only occur within critical sections, access to which need to be arbitrated using non-trivial resource-sharing protocols.

In light of these benefits and drawbacks of preemption, *limited-preemption scheduling* has emerged as an attractive alternative to the extremes of preemptive and non-preemptive scheduling. Under limited-preemption scheduling a process may only be preempted at specified "potential preemption points" — see Figure 1 — that are designated as being so by the software engineering team developing the process; the run-time environment is required to let the process execute non-preemptively between successive preemption points. These potential preemption points are presumably designated at points in the code where the cost of preemption are somewhat limited. Schedulability analysis techniques such as Response-time Analysis [4, 13] or the Processor Demand Methodology [6, 7, 26] have been adapted to allow for such non-preemptive execution: in such extended analysis the maximum duration of such non-preemptive execution is typically represented as a *blocking time* (so called because it represents the duration for which an executing process may block the execution of other processes of greater priority). Hence the blocking time becomes a parameter in the model of the timing behavior of the code (along with the other parameters such as worst-case execution time, deadline, etc.)

Schedulability analysis such as Response-Time Analysis and the Processor Demand Methodology are performed prior to run-time,

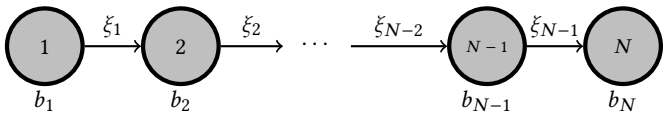

Figure 1: A program is a sequence of non-preemptive basic blocks $1, 2, \ldots, N$, each of which is characterized by a worst-case execution time $b_1, b_2, \ldots, b_N$. Preemptions may occur between basic blocks: ξ_i is the worst-case cost of performing a preemption between the i'th and the $(i+1)$'th basic block.

when the actual execution times and blocking times are unknown; instead, *upper bounds* must be estimated and used. The upper bound on the blocking time that is estimated for a process may in general depend upon which of the potential preemption points actually experience preemptions during run-time. Algorithms have been developed for selecting those of the potential preemption points of a process at which preemptions should occur in order to optimize for certain metrics; e.g., Bertogna et al. [8] present an algorithm for determining which potential preemption points are actually permitted to experience preemptions (these are called *effective preemption points* in [8]) during run-time in order to minimize the overall worst-case execution time of the process, subject to the constraint that the blocking time not exceed a specified value. These algorithms require that estimates be made on the upper bounds of

(1) the execution times of the blocks of code between successive potential preemption points (known as the *worst-case execution times* or *WCET's* of these blocks of code — they are depicted as the b_i values in Figure 1); and
(2) the preemption overhead that would be incurred were an actual preemption to occur at each potential preemption point (these upper bounds are called the *worst-case preemption delays* of these potential preemption points, and are depicted as the ξ_i values in Figure 1).

These upper bounds (i.e., the b_i and ξ_i values in the notation of Figure 1) are provided as inputs to the algorithms selecting the effective preemption points from amongst the designated potential preemption points; given such inputs, the algorithms determine both the effective preemption points and an upper bound on the worst-case running time of the program. Correctness considerations require that input b_i and ξ_i values be *safe* upper bounds in the sense that they bound the actual execution/ preemption times that may be experienced during run-time; otherwise the outputs of the algorithms may not be valid (e.g., the actual blocking time that is experienced during run-time may exceed the specified value, or the actual running time of the process may exceed the bound that is computed by the algorithm). Modern computing platforms are characterized by significant variation and unpredictability with regards to timing behavior:

- the same piece of code executing upon the same platform may require very different durations of time to execute on different runs, and
- preemption delay typically depends not just upon the characteristics of the process being preempted but also upon the characteristics of other processes with which this process exists concurrently (and which therefore preempt it or are preempted by it).

As a consequence, safe upper bounds on the b_i and ξ_i values tend to be very conservative over-estimations of the execution time and preemption delay that is typically experienced under most circumstances.

This research. The major research question investigated in this paper is this: if we had access to not just worst-case bounds on the execution time of the basic blocks and preemption delay parameters, but also *typical* values of these parameters, could we use these additional estimates to our benefit? We answer this question in the affirmative: we show that careful and considered use of such typical execution-time/ delay parameter values, if available to us, permits us to optimize for a wider range of run-time properties while continuing to guarantee that worst-case blocking will not exceed a specified bound. Specifically, recall that Bertogna et al. [8] have derived an algorithm that determines, for a given program, the choice of effective preemption points that minimizes its overall worst-case execution time plus any cache-related preemption delay subject to the constraint that blocking time (i.e., the maximum non-preemptive region for a task) not exceed a specified value, and provides an upper bound on the resulting worst-case running time (execution including cache reloads) for the program. We show how effective preemption points may be chosen to minimize the *typical*, rather than worst-case, overall running time of the process (subject to the same constraint that the blocking time not exceed a specified value, even in the worst case), and how to obtain a safe estimate on the resulting worst-case running time.[1][2] We consider two variations: one in which our only concern is the typical running time and hence there are no restrictions placed on the worst-case behavior, and the other in which we must satisfy an additional constraint that the worst-case running time of the process also not exceed a specified threshold value. We consider both off-line and on-line versions of these problems:

- In the *off-line* versions, the effective preemption points are to be determined beforehand: during run-time each process will be preempted at all such pre-determined effective preemption points.
- In the *on-line* versions, in contrast, the run-time behavior of the process – the actual durations for which code has executed and the actual delays that already-occurred preemption have experienced – is used to help determine future effective preemption points.

Organization. The remainder of this paper is organized in the following manner. In Section 2 we describe the formal workload model that we use in this paper to represent the timing requirements of preemptable real-time code. We discuss models that have previously been used, and motivate and formally define the generalizations that we are proposing to these prior models. Our main technical results are presented in Sections 3 and 4; in Section 3 we present our results concerning the off-line version of the problem when all preemption points are required to be selected prior to beginning the execution of the code, and in Section 4 we present our results for the on-line version where some of the preemption points are selected during run-time once the code has begun executing. We provide a brief survey of some of the vast body of related research on limited-preemption scheduling in Section 5, and conclude in Section 6 with a brief summary of our results and a discussion regarding future directions.

[1] That is, we provide an upper bound on the running time of the program under the assumption that no duration — of either a preemption or the execution of a basic block — exceeds the estimate of its typical value.

[2] We do not address in this paper how typical runtimes for basic blocks/preemptions and leave it to determine the mechanism for setting these values. For instance, the designer might follow the strategy of [19] take several traces of execution of the tasks with various preemption points and use the maximum observed runtime as input; this maximum observed value is guaranteed to be less pessimistic than the worst-case runtimes derived using static analysis.

2 MODEL, AND PRIOR RESULTS

In this section we formally present the workload model that was informally discussed in Section 1 above (Section 2.1), briefly discuss some prior results that are relevant to the reminder of this paper (Section 2.2), and describe the extensions to the prior model that we are proposing (Section 2.3).

2.1 Models used in prior work

Bertogna et al. [8] model a program as a sequence of N non-preemptive *basic blocks* (BB's) — see Figure 1. Each basic block is characterized by a worst-case execution time (WCET); we denote the WCET of the i'th block as b_i, $1 \leq i \leq N$. Preemption is permitted only at the boundaries between basic blocks (conditional branches and critical sections are assumed to be executed entirely within a basic block); hence there is a *potential preemption point* between successive basic blocks. There is a worst-case preemption cost associated with performing a preemption at a potential preemption point; we denote the worst-case preemption cost of performing a preemption between the i'th and $(i+1)$'th basic block as ξ_i.

It is generally not required that a program be preempted at all its potential preemption points during run-time – the ones at which preemptions will actually occur are called the *effective preemption points*. The sequence of basic blocks between any pair of consecutive effective preemption points is referred to as a *non-preemptive region*. Suppose that a non-preemptive region comprises the i'th through the j'th basic block ($1 \leq i \leq j \leq N$). It is evident that this non-preemptive region, along with the preemption that was necessary prior to its execution, executes non-preemptively during run-time for a duration that may be as large as $\left(\xi_{i-1} + \sum_{\ell=i}^{j} b_\ell\right)$. (Here we are adopting the convention that $\xi_0 \leftarrow 0$.) The worst-case running time of the entire program is equal to the sum of these durations over all the non-preemptive regions of the program.

2.2 Relevant prior results

The following problem was studied in [8]:

Given a program specified via the equi-sized vectors $\overrightarrow{b} \stackrel{\text{def}}{=} [b_1, b_2, \ldots, b_N]$ of basic block WCETs and $\overrightarrow{\xi} \stackrel{\text{def}}{=} [\xi_o \equiv 0, \xi_1, \xi_2, \ldots, \xi_{N-1}]$ of worst-case preemption costs, and a blocking parameter Q denoting an upper bound on the maximum duration for which a program may execute non-preemptively,

Determine the minimum possible worst-case running time that can be guaranteed for this program (subject to the constraint that no non-preemptive region has a run-time greater than Q), and a selection of effective preemption points that guarantees this minimum worst-case running time. □

Bertogna et al. [8] derived an algorithm based on dynamic programming for solving this problem, which we now describe. Let $B(k)$ denote the smallest possible worst-case running time, including preemption overhead costs, that is incurred in executing the first k basic blocks. (The desired solution to the problem is therefore the value of $B(N)$ and the selection of effective preemption points that yields this value). Setting

$$B(0) = 0 \tag{1}$$

Bertogna et al. [8] observed that for any $k \geq 1$, the value of $B(k)$ may be expressed in terms of the values of the $B(\ell)$'s for $\ell < k$, in the following manner:

$$B(k) = \min_{j \in \mathbb{J}} \left\{ B(j) + \xi_j + \sum_{\ell=j+1}^{k} b_\ell \right\} \tag{2}$$

with the j's constrained as follows:

$$\mathbb{J} = \left\{ j \ : (0 \leq j < k) \text{ and } \left(\xi_j + \sum_{\ell=j+1}^{k} b_\ell\right) \leq Q \right\} \tag{3}$$

This recurrence may be understood based on the following reasoning. Suppose that the last preemption before the execution of the k'th basic block occurred between the j'th and $(j+1)$'th basic blocks. Since the maximum duration of this preemption is ξ_j, and $\left(\sum_{\ell=j+1}^{k} b_\ell\right)$ represents the cumulative WCETs of all the blocks that have executed since that last preemption, the sum $\left(\xi_j + \sum_{\ell=j+1}^{k} b_\ell\right)$ represents the maximum duration for which the process may have executed non-preemptively since this last preemption. Hence, $\mathbb{J}$ denotes (the indices of) the set of all potential preemption points at which the process may have been preempted for the last time prior to the execution of the k'th basic block, without violating the constraint that no non-preemptive region have a run-time greater than Q. The smallest possible worst-case running time incurred in executing the first k basic blocks is now determined by simply computing the minimum of the worst-case running times if the last preemption were to occur at each of these possible preemption points in $\mathbb{J}$.

While determining $B(k)$ according to Expression 2 above, we additionally store in prev(k) the value of $j \in \mathbb{J}$ that defines the value of $B(k)$; i.e., the value of j at which the RHS of Expression 2 is minimized. Once $B(N)$ has been determined, the sequence of effective preemption points is determined as prev(N), prev(prev(N)), prev(prev((N))), . . ., all the way down until 0.

To illustrate the problem and the approach of Bertogna et al. [8] consider the program depicted in Figure 2 represented by two vectors $\overrightarrow{b} = [b_1, b_2, b_3] = [5, 3, 4]$ and $\overrightarrow{\xi} = [\xi_0, \xi_1, \xi_2] = [0, 3, 2]$, and with $Q = 10$. Two potential preemption points are specified; hence there are 2^2 or four possible choices for effective preemption points. It turns out that having a single effective preemption point, between basic blocks 2 and 3, is the choice that results in the smallest overall worst-case running time (which equals $5 + 3 + 2 + 4 = \mathbf{14}$).

2.3 Our proposed model extensions

As mentioned in Section 1, the major research question we seek to investigate is this: if we had access to not just worst-case bounds on the execution time and preemption delay parameters, but also *typical* values of these parameters, what use could we make of these additional estimates? (For instance, the worst-case bounds could be obtained by doing conservative path and cache-eviction analysis, while the typical values could be measurement-based.) We therefore continue to model our program as before, as a sequence of N non-preemptive basic blocks, but suppose that we have two estimates for the execution time of each basic block and the preemption delay associated with each potential preemption,

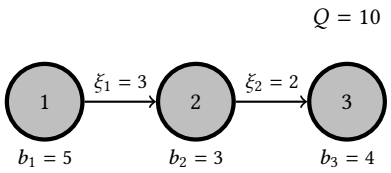

Figure 2: An example. The algorithm of [8] would yield $B(1) = 5, B(2) = \min(B(0) + \xi_0 + b_1 + b_2, B(1) + \xi_1 + b_2) = \min(0 + 0 + 5 + 3, 5 + 3 + 3) = 8$, and $B(3) = \min(B(1) + \xi_1 + b_2 + b_3, B(2) + \xi_2 + b_3) = \min(5+3+3+4, 8+2+4) = 14$ on this example. Note that for $B(3)$ the scenario where only ξ_0 is selected (i.e., the code executes non-preemptively) is not considered in the $\min()$ calculation since the non-preemptive execution of the entire code would exceed the non-preemptive region constraint Q; hence the selected effective preemption point is the one between basic blocks 2 and 3.

a "worst-case" one and a "typical" one. The worst-case and typical estimates for the execution time of the i'th basic block are denoted as $b_i^{(W)}$ and $b_i^{(T)}$ respectively, and the worst-case and typical estimates for the cost of performing a preemption between the i'th and the $(i+1)$'th basic blocks are denoted as $\xi_i^{(W)}$ and $\xi_i^{(T)}$ respectively. Hence in our more general model a program is specified via the four vectors $\overrightarrow{b^{(T)}} \stackrel{\text{def}}{=} [b_1^{(T)}, b_2^{(T)}, \ldots, b_N^{(T)}]$, $\overrightarrow{b^{(W)}} \stackrel{\text{def}}{=} [b_1^{(W)}, b_2^{(W)}, \ldots, b_N^{(W)}]$, $\overrightarrow{\xi^{(T)}} \stackrel{\text{def}}{=} [\xi_0^{(T)} = 0, \xi_1^{(T)}, \xi_2^{(T)}, \ldots, \xi_{N-1}^{(T)}]$, and $\overrightarrow{\xi^{(W)}} \stackrel{\text{def}}{=} [\xi_0^{(W)} = 0, \xi_1^{(W)}, \xi_2^{(W)}, \ldots, \xi_{N-1}^{(W)}]$. As in earlier work [8], we will also specify the bound Q on the maximum duration for which any a non-preemptive region is permitted to execute. For those versions of the problem we consider here in which an upper bound is specified for the worst-case running time of the program, we let D denote the specified upper bound.

Throughout this paper we assume that all the parameters are integers, and that $b_k^{(W)} > 0$ for all k (since a basic block with worst-case execution time equal to zero may simply be merged with its predecessor or successor basic block). We also assume that $\xi_i^{(W)} + b_{i+1}^{(W)} \leq Q$ for all i, since otherwise the presence of this potential preemption point is meaningless: we can never risk preempting between basic blocks i and $(i+1)$. (If this condition is not satisfied for some i, then we may merge the i'th and $(i+1)$'th basic blocks into a single basic block as a pre-processing step upon the input program.)

The problem considered. Given a program specified as above, we consider the problem of selecting a subset of the potential preemption points to designate as being actual preemption points in order to minimize the typical running time of the program (subject to the same constraint that the blocking time is bounded from above by the specified constant Q even in the worst case). That is, *our **objective** is to determine the smallest overall running time that can be guaranteed for the program provided that no duration, of either a preemption or a basic block execution, "overruns" or exceeds its estimated typical value, and to specify the selection of effective preemption points that guarantees this value.* We emphasize that we are seeking not only to minimize the overall running time under typical conditions; we must also be able to predict, prior to run-time, what this overall running time under typical conditions is going to be — such pre-runtime predictability is an essential requirement.[3]

3 SELECTING EFFECTIVE PREEMPTION POINTS PRIOR TO RUN-TIME

Bertogna et al. [8] have obtained an algorithm, briefly described in Section 2.2 above, for determining which of the potential preemption points of a program should be designated as effective preemption points in order to minimize the worst-case running time of the program, subject to the constraint that no non-preemptive region have a worst-case running time greater than a specified blocking bound Q. We now consider the selection of effective preemption points in order to minimize the typical running time of the program; we first consider, in Section 3.1, this problem when no constraint is placed upon the worst-case overall running time of the program; subsequently in Section 3.2 we consider the same problem when an upper bound D on the worst-case running time of the program is also specified.

3.1 No bounds on worst-case running time

Let us consider a program specified by the four equi-sized vectors $\overrightarrow{b^{(T)}}$, $\overrightarrow{b^{(W)}}$, $\overrightarrow{\xi^{(T)}}$ and $\overrightarrow{\xi^{(W)}}$ as discussed in Section 2.3 above, and let Q denote the maximum permitted duration of a non-preemptive block of execution. Analogous to Section 2.2, let $B(k)$ denote the minimum possible completion time for the k'th basic block if each basic block were to execute for a duration no larger than its typical execution time and each preemption were to take a duration no larger than its typical preemption duration. As in [8], we can express the value of $B(k)$ as a recurrence in terms of values of $B(\ell)$'s for $\ell < k$. The critical observation in constructing the recurrence is that we should define the value of $B(k)$ using the *typical* values (the $\xi_i^{(T)}$'s and $b_i^{(T)}$'s) rather than the worst-case values (the $\xi^{(W)}$'s and $b_i^{(W)}$'s):

$$B(k) = \min_{j \in \mathbb{J}} \left\{ B(j) + \xi_j^{(T)} + \sum_{\ell=j+1}^{k} b_\ell^{(T)} \right\} \tag{4}$$

Note, however, that *the values of j that belong to $\mathbb{J}$ should continue to be defined based on the worst-case values (the $\xi_i^{(W)}$'s and $b_i^{(W)}$'s)*, in order to ensure that the non-preemptive execution duration is $\leq Q$ not just in the typical case but also in the worst case:

$$\mathbb{J} = \left\{ j \ : (0 \leq j < k) \text{ and } (\xi_j^{(W)} + \sum_{\ell=j+1}^{k} b_\ell^{(W)}) \leq Q \right\} \tag{5}$$

(As in Section 2.2, we will store in $\text{prev}(k)$ the value of $j \in \mathbb{J}$ that defines the value of $B(k)$; once $B(N)$ is computed, these values of $\text{prev}(\cdot)$ can be used to identify the effective preemption points.)

[3] One consequence of this requirement that (an upper bound on) the running time of the program under typical conditions be *a priori* predictable is that we make no effort to exploit differences between actual durations of execution/ preemption and their estimated typical values. That is, we choose to not do any run-time "slack reclamation" — such reclamation complicates the run-time algorithm without impacting the *a priori* predictions.

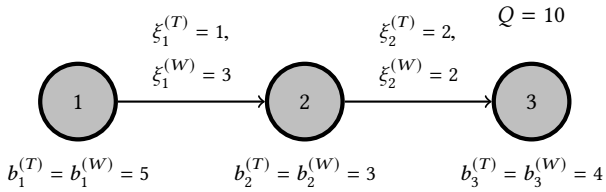

Figure 3: Minimize typical running time (i.e., preempt between basic blocks 1 and 2).

Example 1. *Consider the program depicted in Figure 3. Notice that the worst-case characterization of this program yields exactly the program depicted in Figure 2; as we have stated in Section 2.2, on this example the algorithm of [8] identifies an effective preemption point between basic blocks* 2 *and* 3 *(and a corresponding worst-case running time of* 14*). We can see that with this effective preemption point, the typical running time is also* 14*:*

$$b_1^{(T)} + b_2^{(T)} + \xi_2^{(T)} + b_3^{(T)} = 5 + 3 + 2 + 4 = \mathbf{14}$$

However, choosing instead the potential preemption point between basic blocks 1 *and* 2 *would have resulted in a typical running time of* 13 *while continuing to respect the constraint that blocking time by* ≤ 10 *even in the worst case. This is the solution returned by applying the recurrence algorithm implied by Expressions (4) and (5) rather than the algorithm of [8]. It may be verified that the values of* $B(0), B(1),$ *and* $B(2)$ *are identical; the value of* $B(3)$ *is computed as follows:*

- *As stated above, the set* $\mathbb{J}$ *is computed using the worst-case parameter estimates; hence all three basic blocks cannot be placed in the same non-preemptive region (i.e.,* $0 \notin \mathbb{J}$*), and we thus have* $\mathbb{J} = \{1, 2\}$
- *We consequently have*

$$\begin{aligned} B(3) &= \min\left(B(1) + \xi_1^{(T)} + \left(b_2^{(T)} + b_3^{(T)}\right),\ B(2) + \xi_2^{(T)} + b_3^{(T)}\right) \\ &= \min(5 + \mathbf{1} + 3 + 4,\ 8 + \mathbf{2} + 4) = \min(13, 14) \\ &= \mathbf{13}, \quad \textit{and } \mathrm{prev}(3) = 1. \end{aligned}$$

We therefore conclude that the smallest value of the typical running time is 13*; to achieve this, we should choose as the effective preemption points the potential preemption points (i)* prev(3) *which equals* 1*, and (ii)* prev(1) *which equals* 0*. That is, we should choose the potential preemption point between basic blocks* 1 *and* 2 *as the effective preemption point.* □

3.2 Worst-case running time bounded from above

Although the algorithm derived in Section 3.1 results in the smallest value of typical running time for a given program and a given bound Q, it pays no attention at all to the worst-case running time that the program may experience. We now consider a variant of the problem: it is additionally required that the worst-case running time of the program not exceed a specified upper bound D. Given such a bound, we seek to determine which of the potential preemption points of the program should be designated as actual preemption points in order to minimize the typical running time, subject to the constraints that no individual non-preemptive block have a worst-case duration exceeding Q and the worst-case running time of the entire program not exceed D. We have two major results concerning this problem:

(1) We first show, in Section 3.2.1 below, that this problem is NP-hard; hence, we are unlikely to be able to find a polynomial-time algorithm for solving it.
(2) In Section 3.2.2, we present a pseudo-polynomial time algorithm that has run time polynomial in the value of D, the upper bound on the worst-case running time of the program. It is thus fairly efficient for instances where the specified value of D is not too large.

3.2.1 NP-hardness. In this section we will show that the *decision version* of our problem —given a program specified as described in Section 2.3, a bound D on the worst-case running time of the program, and a positive integer T, is there a placement of effective preemption points in the program such that the worst-case running time is at most D and the typical running time is at most T— is NP-hard by transforming from the PARTITION problem, which is defined as follows:

Given a (multi)set $S = \{a_1, a_2, \ldots, a_n\}$ of positive integers with $A \stackrel{\text{def}}{=} \left(\sum_{i=1}^{n} a_i\right)/2$, ***determine*** whether there is an $S' \subseteq S$ for which $\sum_{a_i \in S'} a_i = A$.

PARTITION was shown to be NP-hard in [14]. It is straightforward to show that PARTITION remains NP-hard even if the set S is required to satisfy the additional restriction that each element in S is strictly smaller than $A/2$ (i.e., each element in S is smaller than a quarter of the sum of all the elements). Henceforth in this section we will therefore assume that $a_{\max} \stackrel{\text{def}}{=} \max_{a_i \in S}\{a_i\}$ is strictly smaller than $A/2$. Furthermore, this restriction and the fact that element sizes must be positive integers implies that A exceeds 2.

Given a set S of positive integers with each smaller than a quarter of the sum of all the elements in S, we now describe a procedure to obtain

(1) a program specified by the four equi-sized vectors $\overrightarrow{b^{(T)}}$, $\overrightarrow{b^{(W)}}$, $\overrightarrow{\xi^{(T)}}$, and $\overrightarrow{\xi^{(W)}}$,
(2) a value for Q,
(3) a value for D, and
(4) a value for T

such that effective preemption points can be selected for the program to ensure a blocking time at most Q, worst-case running time at most D, and typical running time T, if and only if $S \in$ PARTITION.

We now describe this procedure. Recall that A denotes half the sum of the n elements in S: $\sum_{i=1}^{n} a_i = 2A$. The values for Q, D, and T are as follows:

$$Q = \left(\frac{3}{2}A - 1\right) \tag{6}$$

$$D = \left(\frac{3}{2}A + Q\right) \times n + 3A \tag{7}$$

$$T = \left(\frac{3}{2}A + Q\right) \times n + A \tag{8}$$

The program is a sequence of $4n$ basic blocks, obtained by concatenating the n instantiations, for $i = 1, 2, \ldots, n$, of the "gadget" of Figure 4. Intuitively, the i'th instantiation of this "gadget"corresponds to the element a_i of S. We now argue that of the two potential preemption points — the one between basic blocks $(4i - 3)$

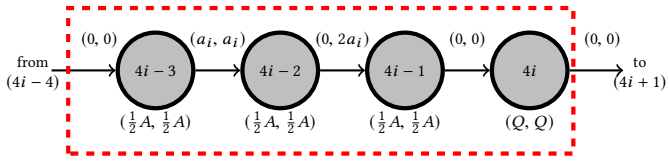

Figure 4: "Gadget" used in the NP-hardness reduction of Section 3.2.1

and $(4i-2)$, and the one between basic blocks $(4i-2)$ and $(4i-1)$ — at least one must be designated an effective preemption point:

- The basic block $4i$ has a worst-case execution time equal to Q, and must therefore execute as its own non-preemptive block. Hence, the potential preemption points between basic blocks $(4i-4)$ and $(4i-3)$, between basic blocks $(4i-1)$ and $4i$, and between basic blocks $4i$ and $(4i+1)$ *must* be designated as effective preemption points.
- The three basic blocks $(4i-3)$, $(4i-2)$, and $(4i-1)$ together have a worst-case execution time of $\frac{3}{2}A$, which exceeds Q; hence, we must preempt at least once between these three blocks.

It is evident that it is not necessary to preempt at both of these two potential preemption points; it remains to compare the consequences of choosing between the two:

(1) If we were to preempt between basic blocks $(4i-3)$ and $(4i-2)$, both the typical duration and the worst-case duration would be
$$\left\lfloor \frac{1}{2}A \right\rfloor + \left\lfloor a_i + \frac{1}{2}A + \frac{1}{2}A \right\rfloor = \left\lfloor \frac{1}{2}A \right\rfloor + \lfloor a_i + A \rfloor = \left(\mathbf{a_i + \frac{3}{2}A}\right)$$
— here the terms within the square parentheses ([ ]) denote the execution durations of the resulting non-preemptive regions. Note that since both $\left(\frac{1}{2}A \leq \frac{3}{2}A - 1\right)$ and
$$a_i + A \leq a_{\max} + A < A/2 + A,$$
each such execution duration is no larger than Q, as required.

(2) If we were to preempt between basic blocks $(4i-2)$ and $(4i-1)$, the typical duration would be
$$\left\lfloor \frac{1}{2}A + \frac{1}{2}A \right\rfloor + \left\lfloor 0 + \frac{1}{2}A \right\rfloor = [A] + \left\lfloor \frac{1}{2}A \right\rfloor = \left(\mathbf{\frac{3}{2}A}\right)$$
while the worst-case duration would be
$$\left\lfloor \frac{1}{2}A + \frac{1}{2}A \right\rfloor + \left\lfloor 2a_i + \frac{1}{2}A \right\rfloor = [A] + \left\lfloor 2a_i + \frac{1}{2}A \right\rfloor = \left(\mathbf{2a_i + \frac{3}{2}A}\right)$$
Note again that since both $\left(A < \frac{3}{2}A - 1\right)$ and
$$2a_i + \frac{1}{2}A \leq 2a_{\max} + \frac{1}{2}A < \frac{3}{2}A,$$
each non-preemptive execution duration is no larger than Q.

We thus see that the "gadget" of Figure 4 corresponding to the element $a_i \in S$ can have effective preemption points inserted to have *either* both typical-case and worst-case running times equal to $\left(Q + \frac{3}{2}A + a_i\right)$, *or* typical-case running time $\left(Q + \frac{3}{2}A\right)$ and worst-case running time $\left(Q + \frac{3}{2}A + 2a_i\right)$. Each gadget therefore contributes at least $\left(Q + \frac{3}{2}A + a_i\right)$ to the worst-case running time; some – those in which the effective preemption point is selected to between basic blocks $(4i-2)$ and $(4i-1)$ – contribute an additional a_i. Let $\mathbb{I}$ denote those values of i for which the effective preemption point is selected to between basic blocks $(4i-2)$ and $(4i-1)$, and the gadgets corresponding to which hence contribute an additional a_i term to the worst-case running time. Summing over all n gadgets, we have

Total worst-case running time
$$\begin{aligned} &= \sum_{i=1}^{n}\left(Q + \frac{3}{2}A + a_i\right) + \sum_{i\in\mathbb{I}} a_i \\ &= \left(Q + \frac{3}{2}A\right) \times n + \sum_{i=1}^{n} a_i + \sum_{i\in\mathbb{I}} a_i \\ &= \left(Q + \frac{3}{2}A\right) \times n + 2A + \sum_{i\in\mathbb{I}} a_i \end{aligned} \tag{9}$$

The last step follows from $2A = \sum_{i=1}^{n} a_i$ by definition of A. Observe that the contribution to the typical run-time of the gadget corresponding to a_i is equal to $\left(Q + \frac{3}{2}A\right)$ for each $i \in \mathbb{I}$, and $\left(Q + \frac{3}{2}A + a_i\right)$ for each $i \notin \mathbb{I}$. Summing over all n gadgets, we have

Total typical running time
$$= \sum_{i=1}^{n}\left(Q + \frac{3}{2}A\right) + \sum_{i\notin\mathbb{I}} a_i = \left(Q + \frac{3}{2}A\right) \times n + \sum_{i\notin\mathbb{I}} a_i \tag{10}$$

It follows from Expressions 9 and 10 above that we can choose effective preemption points to have a worst-case running time no larger than $\left(\left(Q + \frac{3}{2}A\right) \times n + 3A\right)$, which is the value of D specified in Expression 7, *and* a typical running time no larger than $\left(\left(Q + \frac{3}{2}A\right) \times n + A\right)$ —the value of T specified in Expression 8— if and only if we have both $\left(\sum_{i\in\mathbb{I}} a_i \leq A\right)$ and $\left(\sum_{i\notin\mathbb{I}} a_i \leq A\right)$. It is evident that this is possible if and only if the set S can be partitioned into two subsets each of which sums to A; i.e., the set $S \in$ PARTITION.

3.2.2 An algorithm with pseudo-polynomial run-time. In Section 3.2.1 above we showed that the problem of selecting effective preemption points to minimize typical running times subject to the additional constraint that the worst-case running time not exceed a specified value is NP-hard, by reducing from the PARTITION problem. Although this transformation implies that we cannot solve our problem with a polynomial-time algorithm (under the widely-held assumption that P ≠ NP), it does not rule out the possibility of nevertheless obtaining reasonably efficient algorithms, albeit not polynomial-time ones, for solving it — indeed, Hayes[12] refers to PARTITION as the "easiest hard problem". We will now derive an algorithm for solving our problem that has running time polynomial in the value of D; since one typically does not specify very large upper bounds, it is realistic to expect that the value of D would generally be reasonably small.

We start out with some notation: let $B(k, w)$ denote the minimum typical running time that can be obtained for the first k basic blocks, *subject to the constraint that the worst-case running time in executing these k basic blocks not exceed w*. Our objective is to determine the effective preemption points that result in the value of $B(N, D)$. First

note from the definition of the $B(\cdot,\cdot)$ function that

$$B(0,\, w) = 0 \text{ for all } w \geq 0 \tag{11}$$

that is, the typical running time for the (dummy) basic block 0 is always zero, and

$$B(k,\, w) = \infty \text{ for all } k \geq 1 \text{ and } w \leq 0 \tag{12}$$

that is, we cannot have a finite typical running time for the non-empty sequence of basic blocks $1, 2, \ldots, k$ if the worst-case running time is not permitted to be non-negative (we are using the fact here that $b_1^{(W)} \geq 1$).

For $k > 0$ and $w > 0$, we can write a recurrence expressing the value of $B(k, w)$ in terms of the values of $B(\ell, \omega)$ with $\ell < k$ and $\omega < w$, as follows:

$$B(k,\, w) = \min_{j \in \mathbb{J}} \left\{ B\Big(j,\, w - \big(\xi_j^{(W)} + \sum_{\ell=j+1}^{k} b_\ell^{(W)}\big)\Big) + \xi_j^{(T)} + \sum_{\ell=j+1}^{k} b_\ell^{(T)} \right\} \tag{13}$$

with the j's constrained to ensure that the duration of non-preemptive region spanning from a preemption after block j to the end of block k does not exceed the blocking bound Q (even in the worst case):

$$\mathbb{J} = \left\{ j \;:\; \big(\xi_j^{(W)} + \sum_{\ell=j+1}^{k} b_\ell^{(W)}\big) \leq Q \right\} \tag{14}$$

Additionally, we let $\text{prev}(k, w)$ denote the value of $j \in \mathbb{J}$ that obtains the minimum value for $B(k, w)$ in Expression 13.

We can now simply use the recurrence of Expression 13, along with the initial values specified by Equations 11 and 12, to determine the value of $B(N, D)$; this could be done, for instance, using the following procedure whose worst-case run-time computational complexity is clearly polynomial in the values of N and D (and hence pseudo-polynomial):

```
for k = 1 to N
    for w = 1 to D
        Determine B(k, w) and prev(k, w)
        // (Using Equations 11, 12, and 13.)
    end
end
```

4 ADAPTIVE ON-LINE SELECTION OF EFFECTIVE PREEMPTION POINTS

The algorithms we have looked at thus far have been consistent in structure with earlier work on the problem of selecting effective preemption points, in the sense that the effective preemption points are all determined prior to run-time. In this section, we explore an alternative paradigm: if we could *postpone* until run-time the selection of some of the effective preemption points, could we exploit information that is revealed on-line regarding actual preemption durations and the actual execution times of basic blocks in order to obtain smaller typical running times?

Let us first try and understand what benefit such additional information that is revealed on-line could possibly offer us. Recall from Section 2 that *our **objective** is to determine the smallest overall running time that can be guaranteed for the program provided that no duration, of either a preemption or a basic block execution, exceeds its estimated typical value, and to specify the selection of effective preemption points that guarantees this value.* We reiterate that we are seeking the smallest overall running time under typical conditions that we are able to *predict* beforehand; hence, making the choice of effective preemption points on-line is beneficial only if a smaller *a priori* bound on the overall running time of the program under typical conditions can be obtained than if information that is revealed on-line is not exploited. We will see in the remainder of this section that this is indeed the case, obtaining a polynomial-time optimal algorithm regardless of whether an upper bound is specified or not. (We point out that this contrasts with the results in Section 3, where we saw that the optimal *a priori* determination of all effective preemption points is NP-hard when an upper bound is placed on the worst-case running time.)

4.1 An example illustrating the use of an on-line strategy

Let us first consider the illustrative example that is depicted in Figure 5; suppose that a worst-case bound $D \leftarrow 20$ were specified for this example. We start with a few observations on this example:

- Since the sum of the typical execution requirements of all the basic blocks ($b_1^{(T)} + b_2^{(T)} + b_3^{(T)} + b_4^{(T)} = 2+3+3+3 = 11$) exceeds the value of Q (which is 10), at least one preemption may be necessary even under typical conditions.
- Since the strategies of Section 3 specify the effective preemption points statically, they cannot place basic blocks 2, 3, and 4 in the same non-preemptive region since $\xi_2^{(W)} + b_2^{(W)} + b_3^{(W)} + b_4^{(W)} = 2+4+4+3 = 13$, which also exceeds the value of Q. It may be verified that placing basic blocks 1 and 2 in one non-preemptive region and basic blocks 3 and 4 in another (by identifying the potential preemption point between basic blocks 2 and 3 as an effective preemption point) yields a correct solution for which the typical running-time bound is 14 (and the worst-case bound, at 17, is $\leq$ the specified value of 20 for D).

In Figure 6 we present, in pseudo-code form, an on-line strategy for scheduling this example that guarantees a bound of 12 on the running time of the program under typical circumstances (i.e., if no duration, of either preemption or basic-block execution, exceeds its estimated typical value), while also respecting the specified bound of 20 (assuming, of course, that no duration exceeds its worst-case specified value – the $\xi_i^{(W)}$ and $b_i^{(W)}$ values). We will explain later how such strategies are obtained; for now, please notice that

(1) If any preemption/ basic-block execution duration exceeds its typical estimated value (i.e., if any of the **if** clauses evaluates to true), the strategy specifies all the effective preemption points that are to be subsequently chosen for the remainder of the program under the assumption that all subsequent preemptions/ basic-block executions will also exceed their typical values.
(2) The scenarios corresponding to each of the four exit points in this strategy may be verified to be correct in the sense that there is no non-preemptive region (comprising a preemption and all the basic blocks that follow it up to, but not including, the next preemption) is of duration exceeding Q (i.e., 10). It may also be verified that none exceeds the specified bound of 20 on the worst-case running time of the program.
(3) The strategy exits at Line 9 if no preemption/ basic-block execution duration exceeds its typical estimated value; hence

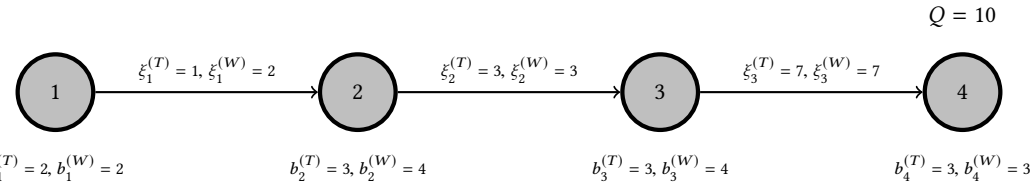

Figure 5: This example is used in Section 4 to illustrate on-line preemption-point selection to minimize the typical running-time bound.

```
1  Execute basic block 1
   // (Since b_1^(T) = b_1^(W), do not consider possible overruns)
2  Preempt; if this preemption takes > 1 time unit
3        Execute basic block 2; preempt;
               execute basic blocks 3 & 4; exit
4  Execute basic block 2; if this execution takes > 2 time units
5        Preempt; execute basic blocks 3 & 4; exit
6  Execute basic block 3; if this execution takes > 3 time units
7        Preempt; execute basic block 4; exit
8  Execute basic block 4
   // (Since b_4^(T) = b_4^(W), do not consider possible overruns)
9  exit
```

Figure 6: On-line strategy for the Example of Figure 5 – worst-case bound $D \leftarrow 20$.

our claim above that the worst-case running time of the program under typical conditions is 12.

4.2 An algorithm for generating on-line strategies

We now describe an algorithm for determining a strategy such as the one in Figure 6 for a given program:

(1) Prior to run-time, our algorithm will identify a set of effective preemption points from amongst the potential preemption points in the program. (There is just one such identified in the strategy of Figure 6 for our example – the one depicted in Line 2.) These are the preemption points at which preemptions will be carried out during run-time, provided there are no overruns; i.e., durations of all the scheduled preemptions, and of the executions of all the basic blocks do not exceed the typical values specified (the $b_i^{(T)}$ and $\xi_i^{(T)}$ values). They are selected in order to minimize the total typical running time of the program, subject of course to the constraint that the maximum duration of any non-preemptive region never exceed Q even in the worst case. The precise manner in which these are identified is described later in this section.

(2) Our algorithm additionally computes, corresponding to each identified effective preemption point and to each basic block, an *alternative* collection of effective preemption points. If the actual preemption duration or the actual basic block execution duration, overruns (i.e., exceeds the estimated typical value), then the corresponding alternative collection of effective preemption points are the ones that are subsequently enforced.

(3) During run-time, we start out executing the program as pre-computed – i.e., non-preemptively between the identified effective preemption points. While doing so, we monitor the actual duration of time for which each basic block executes, and the duration taken for preemption at each of the selected effective preemption points. If any of these durations exceeds the typical values, we immediately switch to the alternative collection of effective preemption points corresponding to this overrun event.

We now discuss the pre-runtime algorithm, describing both (i) how the effective preemption points that are followed as long as there is no overrun is selected; and (ii) how the alternative collections of effective preemption points corresponding to each overrun event are determined. Let $B(k)$ again denote the completion time of the k'th basic block if each basic block were to execute for a duration equal to its typical execution time and each preemption were to take a duration equal to its typical preemption duration. As before, we can express the value of $B(k)$ as a recurrence in terms of values of $B(\ell)$'s for $\ell < k$:

$$B(k) = \min_{j \in \mathbb{J}} \left\{ B(j) + \xi_j^{(T)} + \sum_{\ell=j+1}^{k} b_\ell^{(T)} \right\} \tag{15}$$

While this is the same recurrence as the one defined as Expression 4 in Section 3.1, the values of j over which the RHS of this expression is minimized (i.e., the elements in $\mathbb{J}$) are different. Recall that in determining whether a particular such j belongs to $\mathbb{J}$, we are in effect asking whether correctness is preserved. For this version of the problem correctness requires that (i) no non-preemptive region execute for a duration greater than Q; and (ii) in the event of an overrun, the overall running time of the program remains bounded by the specified bound D. For correctness if the preemption between basic blocks j and $(j+1)$ is the last preemption prior to basic block k (see Figure 7), the following three conditions must hold:

(1) When the preemption duration as well as the execution durations of all the basic blocks $(j+1), (j+2), \ldots, k$ do not exceed their typical values, we want the duration of non-preemptive execution to not exceed the blocking parameter Q:

$$\left(\xi_j^{(T)} + \sum_{\ell=j+1}^{k} b_\ell^{(T)} \right) \leq Q$$

(2) We must consider the possibility that the preemption duration exceeds the typical value $\xi_j^{(T)}$. If so, we must determine

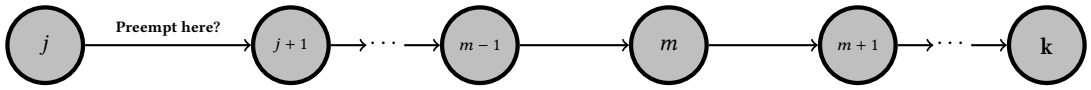

Figure 7: Determining the elements in $\mathbb{J}$: is it safe to preempt?

effective preemption points for the remainder of the program (i.e., from this preemption point onwards) such that

$$B(j) + \mathbb{A} \leq D \tag{16}$$

where $\mathbb{A}$ denotes the *worst-case* running time for the preemption plus the remainder of the program. We explain below how $\mathbb{A}$ is determined.

(3) For each m, $j+1 \leq m \leq k$, we must consider the possibility that basic block m is the first to overrun (i.e., execute for a duration exceeding its typical WCET, $b_m^{(W)}$). In that case, we need that

$$B(j) + \mathbb{B}_m \leq D \tag{17}$$

where $\mathbb{B}_m$ denotes the worst-case execution duration of the preemption plus the basic blocks $(j+1), (j+2), \ldots N$ in this scenario (i.e., with the basic blocks $(j+1), (j+2), \ldots (m-1)$ having already each executed for no more than their estimated typical execution durations). We explain below how the $\mathbb{B}_m$ are determined.

The j's in $\mathbb{J}$ used in Expression 15 are those that satisfy all three conditions enumerated above. And as previously, we continue to let prev(k) denote the value of j that defines the value of $B(k)$ (i.e., that minimizes the RHS of Expression 15). Once $B(N)$ has been computed, these values of prev($\cdot$) can be used to identify the effective preemption points.

Determining $\mathbb{A}$. $\mathbb{A}$ is the smallest worst-case running time for basic blocks $(j+1), \ldots, N$ plus the cost of the preemption between basic blocks j and $(j+1)$. Determining $\mathbb{A}$ is therefore equivalent to determining the smallest worst-case running time of a program consisting of the basic blocks $(j+1), (j+2), \ldots, N$, with the worst-case execution parameter of the $(j+1)$'th basic block inflated to incorporate the cost of the preemption. $\mathbb{A}$ can hence be determined by using the algorithm of [8] (which is described in Section 2.2 of this paper) upon a program specified by the vectors

$$\overrightarrow{b} = \left[\left(\xi_j^{(W)} + b_{j+1}^{(W)}\right), b_{j+2}^{(W)}, \ldots, b_N^{(W)}\right]$$
$$\text{and } \overrightarrow{\xi} = \left[0, \xi_{j+1}^{(W)}, \xi_{j+2}^{(W)}, \ldots, \xi_{N-1}^{(W)}\right]$$

If this value of $\mathbb{A}$ does satisfy Inequality 16 above and the potential preemption point between basic blocks j and $(j+1)$ is selected as an effective preemption point for our strategy, then in the event of the preemption between basic blocks j and $(j+1)$ overrunning (i.e., taking a duration greater than $\xi_j^{(T)}$) during run-time, this output of the algorithm of [8] will constitute the alternative collection of effective preemption points that are used to complete the program's execution with a worst-case execution duration $\leq D$.

Determining $\mathbb{B}_m$. Analogously to the reasoning used above, we argue that $\mathbb{B}_m$ is the worst-case running time for basic blocks $[\hat{m}, m+1, \ldots, N]$, where basic block $\hat{m}$ is an artefact we introduce in order to represent the execution of the preemption along with the basic blocks $(j+1), (j+2), \ldots m$. Note that the preemption, and the basic blocks $(j+1), \ldots, (m-1)$ did <u>not</u> overrun; hence $\hat{m}$ should be assigned a WCET β as follows:

$$\beta \stackrel{\text{def}}{=} \xi_j^{(T)} + \left(\sum_{\ell=j+1}^{m-1} b_\ell^{(T)}\right) + b_m^{(W)}$$

The value of $\mathbb{B}_m$ can now be determined using the algorithm of [8], described in Section 2.2, upon a program specified by the vectors

$$\overrightarrow{b} = \left[(\beta, b_{m+1}^{(W)}, b_{m+2}^{(W)}, \ldots, b_N^{(W)}\right] \text{ and } \overrightarrow{\xi} = \left[0, \xi_m^{(W)}, \xi_{m+1}^{(W)}, \ldots, \xi_{N-1}^{(W)}\right]$$

As with $\mathbb{A}$, if the potential preemption point between basic blocks j and $(j+1)$ is selected as an effective preemption point for our strategy and is additionally the last effective preemption point prior to basic block m, then in the event of the basic block m being the first to overrun (taking a duration greater than $b_m^{(T)}$) during run-time, this output of the algorithm of [8] will constitute the alternative collection of effective preemption points that are used to complete the program's execution with a worst-case execution duration $\leq D$.

Runtime complexity. This approach requires that the actual duration of each preemption and each basic-block execution during program execution be monitored. Assuming this is be done, the additional run-time complexity of implementing the strategy is constant for each preemption and each basic-block execution: one is in essence doing an **if-then-else** with each such duration (see, for instance, the example on-line strategy of Figure 6).

It remains to specify the run-time computational complexity of the pre-runtime algorithm that generates the on-line strategy such as the one in Figure 6. We claim that this is polynomial in the number of basic blocks N in the program: this follows from the observations that

(1) The N values $B(1), B(2), \ldots B(N)$ need to be computed in sequence.
(2) For a given value of k, when computing $B(k)$ we have at most $(k-1)$ candidate values for membership in the corresponding $\mathbb{J}$. Determining whether such a candidate j belongs to $\mathbb{J}$ or not requires us to determine one $\mathbb{A}$, and at most k $\mathbb{B}_m$'s; each such computation requires one call to the polynomial-time algorithm of [8].

4.3 On-line strategies when no bound is specified on the worst-case running time

In Section 3 we considered separately the cases where the objective is to minimize the typical running-time bound when (i) no constraints are placed on the worst-case running time, and (ii) an upper bound D is specified on the worst-case running time. We had seen that while the problem can be solved optimally in polynomial time when no such D is specified, adding the constraint that the worst-case running time of the program also not exceed D renders

the problem NP-hard. For the on-line version of the problem we have seen above (perhaps somewhat counter-intuitively) that even the version with the upper bound D specified is solvable optimally in polynomial time. Hence it is not necessary to separately consider the version of the problem where D is not specified: simply setting the upper bound on worst-case response time equal to infinity ($D \leftarrow \infty$ – in practice, a large number) would yield the required on-line strategy.

5 RELATED WORK

There is a large body of work that focuses upon understanding the effects of preemption upon the execution time of a task. The area of *cache-related preemption delay* (CRPD) analysis (see, e.g., [1–3, 15, 17, 21–23] – this list is by no means exhaustive) seeks to quantify the effect of preemption on existing scheduling approaches. An orthogonal direction of research into reducing the effects of preemption is the area of *limited-preemption scheduling*. In this approach, a task can only be preempted at certain points in its execution. The central goal is to limit the preemption overhead that a job a task experiences during execution. The two different approaches to limited-preemption scheduling are deferred-preemption scheduling and preemption-threshold scheduling. In deferred-preemption scheduling, the preemption due to the arrival of a higher-priority task is delayed until some later time. The length of the delay can either be determined by 1) the maximum blocking time the higher-priority task (called the *floating preemption-point model* [5, 16]), or 2) by pre-determined locations in the task code (called the *fixed preemption-point model* [9]). Preemption-threshold scheduling on the other hand changes the effective priority that task may execute at based upon a preemption threshold which allows a currently-executing lower-priority task to continue executing if a higher-priority task's priority is not greater than some predetermined preemption threshold [25]. Buttazzo et al. [10] provide a survey of limited-preemption scheduling. It is important to note that these and other subsequent works on limited preemption scheduling only attempt to reduce the number of preemptions, but do not explicitly incorporate a quantification of actual CRPD cost nor do they change where the preemption occurs due to potentially different preemption costs at different program locations.

The setting considered in this paper of selecting effective preemption points in a task code combines the benefits of CRPD analysis and limited-preemption scheduling. The CRPD cache-counting approach can be used to quantify the worst-case cost of any preemption; for instance, in our model the $\overrightarrow{\xi^{(W)}}$ vector could be computed by carefully determining the set of cache blocks that would need to be loaded by preempting at each of potential preemption points [11]. Prior research on limited preemption scheduling (e.g., [5]) can be used to determine for each task its maximum blocking parameter Q; that is, this limited-preemption analysis provides the maximum tolerable blocking such that if each task executes any non-preemptive region for no more than Q time units, then the system remains schedulable. Early work in the direction of preemption-point placement provided heuristics for selecting the best points (e.g., Simonson and Patel [20] and Lee et al. [15]). Optimal approaches for preemption placement originated with Bertogna et al. [8] for selecting preemption points for a task's control flowgraph model with a strictly sequential structure (as also assumed in this paper). Later subsequent work, expanded the control flowgraph structure permitted in optimal preemption placement to conditional branches, loops, and function calls [18]. However, these preemption approaches select based upon the worst-case preemption delay that might be realized at a preemption point. Due to the inherent pessimism in CRPD analysis, it is extremely unlikely that this worst-case will always be observed in the actual execution. Thus, in our current work, we have developed an approach that maintains the safety property of the previous approaches (i.e., the effective preemption point chosen will ensure that the task does not execute any non-preemptive region for more than Q time nor exceed any total WCET execution bound of D), but also permits the total execution time to be minimized when more "typical" preemption overheads are experienced.

6 SUMMARY

We have considered the problem of selecting effective preemption points in a program – points at which its execution may be preempted – from amongst a programmer-provided set of potential preemption points in order to minimize the overall preemption overhead (and hence the overall running time) of the program while respecting schedulability constraints. Prior solutions that have been proposed for solving this problem have only considered worst-case characterizations of preemption and code-execution durations; here we have considered a more general model in which both typical estimates and worst-case bounds are assumed available. We have shown how such typical estimates can be used to select effective preemption points in such a manner that we are able to optimally provide minimal *a priori* bounds of the typical overall running time of the program under schedulability constraints that may include upper bounds on both blocking duration and overall worst-case running time. We have further shown that overall running time may be reduced without jeopardizing the schedulability constraints in any manner, by postponing the selection of some of the effective preemption points to run-time (i.e., while the program is executing).

The prime focus of this paper has been the minimization of bounds on the running time of a program under typical conditions, that can be specified prior to run-time. We believe that such work fits in very well with the Vestal-model [24] based mixed-criticality scheduling paradigm: we are in essence specifying two running times –one under typical assumptions and the other, under conservative "worst-case" assumptions– for a single program. We propose to explore the integration of our results and approach with Vestal-based mixed-criticality scheduling theory.

One approach to minimizing running times that we have not considered in this work is dynamic slack reclamation — exploiting differences between predicted (whether worst-case or typical) and actual durations that may be revealed on-line during run-time. As stated in Section 2 (footnote 3), doing so would likely significantly complicate the run-time algorithm without providing any improvement to the *a priori* bounds on typical running time. However, one could envision extensions and generalizations to the particular problem we have studied here in which there is a benefit to on-line slack reclamation of this kind; we plan to study this issue as future work.

ACKNOWLEDGMENTS

This research presented was supported, in part, by the National Science Foundation under Grant Numbers CNS-1618185,CNS-1814739, CNS-1911460, and IIS-1724227.

REFERENCES

[1] S. Altmeyer and C. Burguiere. 2011. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture (JSA), Elsevier* (2011).

[2] S. Altmeyer, R. Davis, and C. Maiza. 2011. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*.

[3] S. Altmeyer, R. Davis, and C. Maiza. 2012. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real Time Systems* 48, 5 (2012).

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. 1991. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*. Atlanta, 127–132.

[5] S. Baruah. 2005. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*.

[6] Sanjoy Baruah. 2006. Resource Sharing in EDF-scheduled Systems: A Closer Look. In *Proceedings of the IEEE Real-time Systems Symposium*. IEEE Computer Society Press, Rio de Janeiro, 379–387.

[7] S. Baruah, A. Mok, and L. Rosier. 1990. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proceedings of the 11th Real-Time Systems Symposium*. IEEE Computer Society Press, Orlando, Florida, 182–190.

[8] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. 2011. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In *2011 23rd Euromicro Conference on Real-Time Systems*. 217–227. https://doi.org/10.1109/ECRTS.2011.28

[9] A. Burns. 1995. *Advances in Real-Time Systems*. Prentice Hall, Inc., Chapter Preemptive priority-based scheduling: an appropriate engineering approach, 225–248.

[10] G. C. Buttazzo, M. Bertogna, and G. Yao. 2013. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics* 9, 1 (2013), 3–15.

[11] J. Cavicchio, C. Tessler, and N. Fisher. 2015. Minimizing Cache Overhead via Loaded Cache Blocks and Preemption Placement. In *Proceedings of the Euromicro Conference on Real-Time Systems*.

[12] Brian Hayes. 2002. Computing Science: The Easiest Hard Problem. *American Scientist* 90, 2 (2002), 113–117. http://www.jstor.org/stable/27857621

[13] M. Joseph and P. Pandya. 1986. Finding Response Times in a Real-Time System. *Comput. J.* 29, 5 (Oct. 1986), 390–395.

[14] R. Karp. 1972. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.). Plenum Press, New York, 85–103.

[15] C.-G. Lee, J. Hahn, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.* 47, 6 (1998), 700–713.

[16] J. M. Marinho, V. Nelis, S.M. Petters, and I. Puaut. 2012. An Improved Preemption Delay Upper Bound for Floating Non-preemptive Region. In *Proceedings of IEEE International Symposium on Industrial Embedded Systems*.

[17] H. S. Negi, T. Mitra, and A. Roychoudhury. 2003. Accurate Estimation of Cache Related Preemption Delay. In *Proceedings of IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (CODES)*.

[18] B. Peng, N. Fisher, and M. Bertogna. 2014. Explicit Preemption Placement for Real-Time Conditional Code. In *Proceedings of Euromicro Conference on Real-Time Systems*.

[19] S. Quinton, M. Hanke, and R. Ernst. 2012. Formal analysis of sporadic overload in real-time systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 515–520.

[20] J. Simonson and J.H. Patel. 1995. Use of preferred preemption points in cache based real-time systems. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*.

[21] J. Staschulat and R. Ernst. 2005. Scalable Precision Cache Analysis for Real-Time Software. *ACM Transactions on Embedded Computing Systems (TECS)* 6, 4 (September 2005).

[22] Y. Tan and V. Mooney. 2004. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proceedings of International Workshop on Software and Compilers for Embedded Systems (SCOPES)*.

[23] H. Tomiyamay and N. D. Dutt. 2000. Program Path Analysis to Bound Cache-Related Preemption Delay in Preemptive Real-Time Systems. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*.

[24] Steve Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press, Tucson, AZ, 239–243.

[25] Y. Wang and M. Saksena. 1999. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real Time Computing Systems and Applications*.

[26] Fengxiang Zhang and Alan Burns. 2009. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Comput.* (2009).