


A Measurement-Based Model for Parallel Real-Time Tasks

Kunal Agrawal¹

Washington University in St. Louis

St. Louis, MO, USA

kunal@wustl.edu


 <https://orcid.org/0000-0001-5882-6647>

Sanjoy Baruah²

Washington University in St. Louis

St. Louis, MO, USA

baruah@wustl.edu

 <https://orcid.org/0000-0002-4541-3445>

Abstract

Under the federated paradigm of multiprocessor scheduling, a set of processors is reserved for the exclusive use of each real-time task. If tasks are characterized very conservatively (as is typical in safety-critical systems), it is likely that most invocations of the task will have computational demand far below the worst-case characterization, and could have been scheduled correctly upon far fewer processors than were assigned to it assuming the worst-case characterization of its run-time behavior. Provided we could safely determine during run-time when all the processors are going to be needed, for the rest of the time the unneeded processors could be idled in low-energy “sleep” mode, or used for executing non-real time work in the background. In this paper we propose a model for representing parallelizable real-time tasks in a manner that permits us to do so. Our model does not require us to have fine-grained knowledge of the internal structure of the code represented by the task; rather, it characterizes each task by a few parameters that are obtained by repeatedly executing the code under different conditions and measuring the run-times.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Real-time schedulability, Theory of computation → Parallel computing models

Keywords and phrases multiprocessor federated scheduling, parallel tasks, work and span, mixed criticality

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.5

1 Introduction

Scheduling theory is concerned with the analysis of real-time systems. As multiprocessor and multicore implementations of real-time systems become prevalent, it is desirable that the models used in scheduling theory for representing real-time workloads be capable of exposing the parallelism that may exist within these workloads. This need has given rise to formal task models such as the *fork-join* model [1, 2], the *sporadic DAG tasks* model [3] (see [4, Chapter 21] for a text-book description), the *multi-DAG* model [5], the *conditional DAG*

¹ Supported in part by NSF grants CCF-1337218 and CCF-173387

² Supported in part by NSF Grants CNS 1409175, CPS 1446631, and CNS 1563845



tasks model [6, 7] etc. Each of these models represents the internal structure of the piece of code being modeled at a relatively fine level of granularity, with the parallelism in the code typically modeled as a directed acyclic graph (DAG). Each vertex in such a DAG represents a segment of sequential code, and edges represent precedence constraints between such code segments: the segment of sequential code represented by the vertex at the tail of an edge must complete execution before the segment of sequential code represented by the vertex at the head of the edge may begin to execute.

Such DAG-based models for representing parallel real-time code have proved popular in the real-time scheduling theory community, and much important and interesting research has been accomplished that is based upon representing systems using these models. This body of research has indeed provided us with a deeper insight into the issues that arise in exploiting parallelism in multiprocessor real-time systems; however due to a variety of reasons (some of which are enumerated and discussed in some detail in Section 2) there are some classes of real-time applications for which such DAG-based representations may not be appropriate for the purposes of schedulability analysis; alternative representations are needed. In this paper we propose one such possible alternative representation that may be suitable under certain circumstances. In this model we do not attempt to explicitly represent the internal parallel structure of the code. Instead, we seek to identify a few important parameters of parallelizable code that are most useful for scheduling algorithms that seek to schedule such code upon multiprocessor platforms, and propose that the code be looked upon as a “black box” that is characterized by just these parameters. Furthermore, we do not require that the internal structure of the code be examined in order to obtain these parameter values. Rather, we propose that values for these parameters be estimated via extensive simulation experiments: repeatedly executing the code in a controlled laboratory environment in order to be able to compute bounds on the parameter values. (Such an approach is inspired by the large and growing body of current research [8] on probabilistic worst-case execution time – pWCET – analysis.) Since measurement-based approaches are typically not able to provide parameter values that are guaranteed correct with absolute certainty, we incorporate, from the mixed-criticality scheduling literature [9], Vestal’s idea [10] of characterizing a single task with two sets of parameters: one set very conservative and hence trusted to a very high level of assurance and the other, far less conservative but more representative of “typical” behavior.

Organization. The remainder of this paper is organized as follows. In Section 2 we motivate the new model by identifying relevant characteristics of parallelizable real-time code that current models are not well-suited to represent, and formally define the workload and system model that we are proposing. In Section 3 we briefly discuss some prior research that provides the foundations upon which our proposed model is built. In Section 4 we derive, and prove the correctness and other relevant properties of, an algorithm for scheduling systems represented using the proposed model. Our overall objective is to be able to obtain more resource-efficient implementations of systems, while ensuring correctness; in Section 5 we explore some possible means of further enhancing the efficiency of the algorithm presented in Section 4. We conclude in Section 6 with a discussion on the relevance, significance, and limitations of our proposed model, and an enumeration of possible directions for continued research.

2 System model: Motivation and Definition

In this section we flesh out the details of the model we are proposing for representing parallelizable real-time code that is not conveniently represented using previously-proposed DAG-based task models. We will first motivate the model informally, and seek to explain aspects of the model via illustrative examples. A formal definition of the model is then provided in Section 2.1; our proposed algorithm for scheduling tasks represented using this model is described in Section 2.2.

Why a new model? As stated in Section 1 above, several excellent DAG-based models for representing parallel real-time code have been developed in the real-time scheduling theory community; however there are some classes of real-time applications for which such models have proved unsuitable. This may be for one or more of the following reasons:

1. The internal structure of the parallel code may be very complex, with multiple conditional dependencies (as may be represented in e.g., the conditional DAG tasks model [6, 7]) and (bounded) loops. Explicit enumeration of all possible paths through such code in order to identify worst-case behavior may be computationally infeasible.³
2. If some parts of the code are procured from outside the application-developers' organization, the provider of this code may seek to protect their intellectual property (IP) by not revealing the internal structure of the code and instead only providing executables – this may be the case if, e.g., commercial vision algorithms are used in a real-time application. (Although reverse-engineering of the executable code in order to determine its internal structure may be possible in principle, such reverse engineering tends to be tedious and error-prone.)
3. Algorithms for the analysis of systems represented using DAG-based models tend to have run-time pseudo-polynomial or exponential in the size of the DAG. Such run-times have traditionally been considered acceptably small enough to allow the algorithms to be practical in practice; however, this state of affairs may not continue in the future. For many cyber-physical real-time systems, constraints such as deadlines are typically dictated by physical factors. As the processors upon which we implement such cyber-physical real-time systems become increasingly more powerful, it becomes possible to incorporate far more complex processing that would be represented as larger DAGs than was previously the case. As this trend towards more complex processing and the consequent larger DAGs continues, run-times pseudo-polynomial in the size of these larger DAGs may become too large to be used in practice during system design and analysis.
4. Further exacerbating the situation, explicitly representing the internal structure of some pieces of parallel code in DAG form results in DAGs that may be of size exponential in the size of the code. Consider, for example, the following code snippet written in OpenMP (<http://www.openmp.org/>), an application programming interface (API) that supports multi-platform shared memory multiprocessing programming:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<10; i++) {
    //do_something
}
```

³ We point out that techniques for *approximating* the worst-case behavior of complex conditional parallelizable code have been proposed with regards to specific scheduling algorithms such as global fixed-priority [6], global EDF [7] or federated [11].

This code snippet would translate to a DAG with $(1 + 10 + 1 =)$ 12 nodes. If we were to replace the “10” in the upper bound of the `for` loop with a “100”, however, the resulting DAG would have 102 nodes; replacing it with “1000” would yield 1002 nodes, etc. – increasing the size of the program by one ASCII character results in an almost ten-fold increase in the size of the DAG.

5. Particularly for conditional code, it may be the case that the true worst-case behavior of the code is very infrequently expressed during run-time.⁴ Traditional models based on conditional DAGs may not be suitable for representing such code (although mixed-criticality [10, 12, 13, 14] extensions of such conditional DAG models are a possibility – to our knowledge, such models have not yet been proposed, let alone studied).

For pieces of parallel real-time code possessing one or more of the characteristics discussed above, DAG-based representations may not be appropriate for the purposes of schedulability analysis; alternative representations are needed. Let us now discuss what such a representation should provide.

Identifying relevant characteristics of parallelizable real-time code. In modeling parallelizable real-time code that is to be executed upon a multiprocessor platform, a prime objective is to enable the exploitation of the parallelism that may be present in the code by scheduling algorithms, in order to enhance the likelihood that we will be able to meet timing constraints. We are interested here in developing *predictable* real-time systems – systems that can have their timing (and other) correctness verified prior to run-time. For the purposes of enabling a priori timing verification, decades of research in the parallel computing community suggests the following two timing parameters of a piece of parallelizable code are particularly significant:

1. The **work** parameter denotes the cumulative worst-case execution time of all the parallel branches that are executed across all processors. Note that for non-conditional parallelizable code this is equal to the worst case execution time of the code on a single processor (ignoring communication overhead from synchronizing processors).
2. The **span** parameter denotes the maximum cumulative worst-case execution time of any sequence of precedence-constrained pieces of code. It represents a lower bound on the duration of time the code would take to execute, regardless of the number of processors available.

The span of a computation is also called the *critical path length* of the computation, and a sequence of precedence-constrained pieces of code with cumulative worst-case execution time equal to the span, a *critical path* through the computation.

The relevance of these two parameters arises from well-known results in scheduling theory concerning the multiprocessor scheduling of precedence-constrained jobs (i.e., DAGs) to minimize makespan – this is the widely-studied $P | \text{prec} | \mathcal{S}_{\max}$ problem in the classic 3-field $\alpha | \beta | \gamma$ notation that is commonly used in scheduling theory [15]. This problem has long been known to be NP-hard in the strong sense [16]; i.e., computationally highly intractable. However, Graham’s *list scheduling* algorithm [17], which constructs a work-conserving schedule by executing at each instant in time an available job, if any are present, upon any available processor, performs fairly well in practice. It was shown [17] that list

⁴ Consider, for example, a real-time application that periodically monitors a sensor for anomalous input. Most of the time the sought-for anomalous input is not detected, and not much computation needs to be performed. But on the rare occasions when anomalous input is detected, considerable additional processing of such input is necessary.

scheduling makes the following guarantee: if \mathcal{S}_{\max} denotes the minimum makespan with which a particular DAG can be scheduled upon m processors, then the schedule generated by list scheduling this DAG upon m processors will have a makespan no greater than $(2 - \frac{1}{m}) \times \mathcal{S}_{\max}$. This result, in conjunction with a hardness result in [18] showing that determining a schedule for this DAG of makespan $\leq \frac{4}{3}\mathcal{S}_{\max}$ remains NP-hard in the strong sense⁵, suggests that list scheduling is a reasonable algorithm to use in practice, and in fact most run-time scheduling algorithms that are used for scheduling DAGs upon multiprocessors use some variant or the other of list scheduling. We will do so in this paper as well.

An upper bound on the makespan of a schedule generated by list scheduling is easily stated. Letting *work* and *span* denote the work and span parameters of the DAG being scheduled, it has been proved in [17] that the makespan of the schedule for a given DAG is guaranteed to be no larger than

$$\frac{\text{work} - \text{span}}{m} + \text{span} \quad (1)$$

Thus a good upper bound on the makespan of the list-scheduling generated schedule for a DAG may be stated in terms of only its work and span parameters. Equivalently if the DAG represents a real-time piece of code characterized by a relative deadline parameter D , $(\frac{\text{work} - \text{span}}{m} + \text{span}) \leq D$ is a sufficient test for determining whether the code will complete by its deadline upon an m -processor platform. *We therefore identify the work and span parameters of a piece of parallel real-time code as being particularly relevant from the perspective of schedulability analysis.*

A measurement-based approach to parameter estimation. The work and span parameters of tasks that are represented using DAG-based models are quite straightforward to compute in time linear in the representation of the DAGs (algorithms for doing so are described in [3, 7]). As we have discussed above, however, our interest is in characterizing parallel code that is typically not conveniently represented using DAG-based models. We propose that for the purposes of representing pieces of such code for schedulability analysis, we *ignore* their internal structure and instead seek to characterize them solely via their work and span parameters. And since we cannot in general determine the precise values of the work and span parameters of a piece of code without knowing its internal structure, we advocate here that *measurement-based* approaches be used to *estimate* these parameters. Measurement-based approaches have been developed for estimating probabilistic worst-case execution time (pWCET) [20, 21, 22] distributions of individual pieces of code, and implemented in pWCET tools such as RapiTime (<https://www.rapitasystems.com/products/rapitime>) from Rapita Systems. We now briefly discuss how measurement-based approaches may be adapted to estimate the work and span parameters of parallel code. Ignoring overhead associated with implementing global scheduling, observe that the work parameter of a piece of code is equal to the time needed to complete its execution upon a single processor, while its span parameter is equal to the time needed to complete its execution upon an unbounded number of processors. Hence one can estimate the probability distribution of the work parameter of a piece of code by using pWCET techniques to estimate its WCET distribution upon a single processor. One can similarly estimate the probability distribution of the span parameter by

⁵ In fact, assuming a reasonable complexity-theoretic conjecture that is somewhat stronger than $P \neq NP$, a result of Svensson [19] implies that a polynomial-time algorithm for determining a schedule of makespan $\leq 2\mathcal{S}_{\max}$ for all m is ruled out.

1. first adapting the measurement-based techniques underpinning pWCET-estimation to determine the makespan probability distribution upon a given number of processors (rather than the completion-time upon a single processor); and then
2. estimating the makespan distribution of the parallel code upon platforms in which the number of processors is repeatedly increased, until further increases do not result in significant changes to the estimated distribution.

The proposed model: multiple work and span estimates. As discussed above, it is possible, by suitable adaptation and application of pWCET techniques, to estimate probability distributions for the work and span parameters of pieces of parallel real-time code. It is understood in the pWCET community that pWCET-based techniques cannot in general determine bounds that are guaranteed to be correct with absolute certainty; rather, they provide bounds that are guaranteed correct to specified probabilistic degrees of confidence/levels of assurance (Davis et al. [8] provide a thoughtful and considered discussion as to how the concept of probabilities should be interpreted when used in such a manner). In the model we propose for representing parallel real-time code, we suggest that each such piece of code be characterized by *two* pairs of (*work*, *span*) parameter values, each pair corresponding to a different probability threshold in the work and span distributions and therefore valid at different levels of assurance. Specifically, one pair of values should be *very* conservative and therefore trusted to a very high level of assurance and the other, while still relatively safe, should be more representative of “typical” behavior, not attempting to cover scenarios that are highly unlikely to occur. We illustrate via an example.

- **Example 1.** Suppose that we were able to determine for a piece of code that
- Its work parameter is > 120 with some small probability p , but > 900 with a far smaller probability $p' \ll p$.
 - Its span parameter is > 40 with probability p ; however the probability that it is > 600 is p' .

We could characterize this piece of code with two ordered pairs of (*work*,*span*) values – a $(1 - p)$ probability of being $\leq (120, 40)$, and a far greater $(1 - p')$ probability of being $\leq (900, 600)$.

We require that *correctness criteria hold under the more conservative estimate*. Suppose for instance that it were specified that this code should execute within a relative deadline of D ; we require that the makespan by $\leq D$ provided *work* ≤ 900 and *span* ≤ 600 . ◀

The proposed run-time scheduling approach. Since correctness is defined with respect to the more conservative estimates for work and span, in order to satisfy correctness requirements we must provision computing resources to a task assuming these more conservative estimates. However, it is our expectation that the task’s run-time behavior is very likely to be bounded by the less conservative parameter estimates, and hence statically provisioning adequate resources for it under the more conservative assumptions is likely to result in significant wastage of computing resources during run-time. One manner of ameliorating such wastage is by keeping some of the provisioned resource in “reserve”, perhaps by placing some processors in sleep mode or having them execute background (non real-time) work, with the option of switching them to work upon executing the task if we determine, during run-time, that the task’s run-time behavior is in fact not likely to be bounded by the less conservative parameter estimates. The following example illustrates.

► **Example 2.** Suppose the code in Example 1 to be scheduled with a relative deadline equal to 690, upon a 10-processor platform. According to Expression 1, the makespan of the schedule upon 10 processors is no more than

$$\frac{900 - 600}{10} + 600 = (30 + 600) = \mathbf{630}$$

assuming the more conservative work and span estimates hold. Hence, correctness is guaranteed.

In fact, ten processors are not necessary for correctness: if we were executing this piece of code upon just four processors, the corresponding makespan bound according to Expression 1 would be $(\frac{900-600}{4}) + 600 = \mathbf{675}$. Our run-time algorithm could therefore realize some energy savings by simply switching off six of the ten provided processors, and executing the task on the remaining four.

Could we switch off seven processors? The reader may verify that with three processors Expression 1 would yield a makespan bound of $(\frac{900-600}{3}) + 600 = \mathbf{700}$. Since 700 exceeds the specified relative deadline of 690, we conclude that we may miss the deadline if we were to switch off seven processors and the system behaved worse than anticipated by its less conservative parameters (although not its more conservative parameters). We therefore conclude that we need at least **4** processors to not be in sleep mode, in order to ensure correctness.

The run-time algorithm we will derive in this paper is designed for systems in which the less conservative parameters are very likely to hold “most of the time”; i.e., the value of p is itself very small. Provided such is the case for this example

- our run-time scheduling algorithm starts out scheduling the system on just three processors, leaving the remaining seven processors in sleep mode.
- If execution has not completed by some time-instant (whose value is precomputed), it wakes up the sleeping processors and makes all ten processors available for this task to execute upon.

We will prove later that with this algorithm, the task completes by the specified deadline of 690 provided the more conservative task parameters hold; **correctness** is thus established. Additionally, there is a $\leq p$ probability that it will not complete by the pre-computed time-instant and hence need to awaken the remaining seven processors.

To evaluate the **efficiency**, suppose, for this example, that $p = 0.05$, indicating that the less conservative parameters hold with 95% probability. There is therefore a $\leq 5\%$ probability that the task will not complete by the specified time-instant,⁶ and the expected number of processors that would be needed is no larger than

$$(0.95 \times 3 + 0.05 \times 10) = (2.85 + 0.5) = \mathbf{3.35}$$

in contrast to the four that would be needed if our run-time scheduling algorithm were to not be used. ◀

Examples 1 and 2 above have illustrated the task model, and the associated run-time strategy for scheduling tasks that are so modeled, that we are proposing in this paper. We now formally define the task model in Section 2.1, and the run-time scheduler in Section 2.2, below.

⁶ In fact, since the work and span parameters will not in general be perfectly correlated, the expected probability of this happening is likely to be far less than 5%. (This issue is revisited in Section 5.)

2.1 System Model

We now provide a formal definition of our model, by describing in detail the workload model we assume. The workload we seek to model comprises a single piece of parallelizable real-time code that is characterized by the following list of parameters:

$$\langle work_O, span_O, work_N, span_N, D \rangle,$$

with the following interpretation:

1. $(work_O, span_O)$. These represent very conservative estimates of the true “worst-case” values of the work and span parameters; as discussed above, we expect that these estimates will be obtained using the kinds of measurement-based techniques that have been developed for estimating probabilistic worst-case execution time (pWCET) distributions.
2. D denotes the *relative deadline* parameter: for correct execution it is required that the job be scheduled with makespan no greater than D .

We highlight here that timing correctness is specified assuming that the $(work_O, span_O)$ parameter estimates are correct: the code is required to complete execution within the specified relative deadline D provided its work and span parameters are no larger than $work_O$ and $span_O$ respectively.

3. $(work_N, span_N)$. These are less conservative estimates on the values of the work and span parameters: it is expected that the actual values of the work and span parameters are *very* likely to be no larger than $work_N$ and $span_N$ respectively. (The subscript “ N ” in $work_N$ and $span_N$ stand for “nominal” [23].)

These parameter estimates play no role in defining correctness; rather (as we have seen in Examples 1 and 2), their values may be used for the purposes of devising more resource-efficient scheduling strategies. It is hence not as critical that their values be assigned correctly as it is for the $work_O$ and $span_O$ parameters: while incorrectly estimated values for $work_O$ and $span_O$ may compromise timing correctness in the sense that we may end up missing deadlines, incorrectly estimated values for $work_N$ and $span_N$ simply result in less efficient implementations.

We assume that $work_N \leq work_O$ and $span_N \leq span_O$. (Although our results are readily extended to situations where these assumptions do not hold, we do not see a rationale for relaxing these assumptions, since by very definition the $(work_N, span_N)$ parameters represent less conservative estimates than the $(work_O, span_O)$ parameters.)

In this paper, we consider the scheduling of a single such task upon a dedicated bank of identical processors. We point out that our results are directly applicable to the scheduling of recurrent – *periodic* or *sporadic* – real-time DAGs under the federated paradigm [24] of multiprocessor scheduling, provided each periodic/ sporadic task satisfies the additional constraint that its relative deadline parameter is no larger than its period parameter (i.e., they are *constrained-deadline* tasks). We believe our approach is particularly appropriate for scheduling systems of recurrent tasks: for such tasks, we anticipate that the $(work_N, span_N)$ parameters will bound the behavior of most invocations (“dag-jobs” [3]) of the task, with an occasional rare dag-job exceeding these bounds. Hence many of the allocated processors will remain in sleep mode most of the time, with the occasional dag-job requiring that the sleeping processors be awakened until that dag-job completes execution, after which they can be returned to sleep mode.

2.2 The Scheduling Algorithm

Given a task as specified above:

$$\langle work_O, span_O, work_N, span_N, D \rangle,$$

that is to be executed upon a platform comprising m identical processors, we first perform some **pre-run-time** schedulability analysis that determines whether we are able to schedule the task upon the m processors in a manner that ensures correctness. Recall that *correctness* is specified as requiring that the task meet its deadline provided its work parameter is $\leq work_O$ and its span parameter is $\leq span_O$: by Expression 1, this is guaranteed provided

$$\left(\frac{work_O - span_O}{m} + span_O \right) \leq D; \quad (2)$$

If Condition 2 does not hold, our scheduling algorithm declares failure: it is unable to schedule this instance in a manner that guarantees timing correctness. Otherwise, it computes a pair of values m_N and \mathcal{S}_N – the manner in which these values are computed will be derived in Section 4.2. These computed parameters have the following intended interpretation: provided the less conservative work and span parameter estimates are correct (i.e., work is $\leq work_N$ and span is $\leq span_N$ for the task), list scheduling can schedule the task upon m_N processors to have a makespan no greater than \mathcal{S}_N .

Run-time scheduling. Suppose that the piece of parallelizable real-time code represented by this task is activated at some time-instant t_o during run-time.

1. The scheduler sets a timer to go off at time-instant $(t_o + \mathcal{S}_N)$, and begins executing the task upon m_N processors using the list-scheduling algorithm [17]. The remaining $(m - m_N)$ processors assigned to this task are placed/ remain in sleep mode.
2. If the task has not completed execution by time-instant $(t_o + \mathcal{S}_N)$, then the scheduler awakens the $(m - m_N)$ sleeping processors, and uses list-scheduling to execute the remainder of the task upon the entire bank of m processors.
3. As mentioned in Section 2.1 above, in the case of recurrent tasks these awakened processors are returned to sleep mode upon completion of execution of the current dag-job of the task.

In Section 4 we will derive the manner in which the values of m_N and \mathcal{S}_N are to be computed in order to guarantee correctness: the algorithm completes execution of the task within D time units of its arrival, provided its work parameter is $\leq work_O$ and its span parameter is $\leq span_O$.

We close this section with an example illustrating the operation of our run-time scheduler.

► **Example 3.** Consider once again the instance discussed in Examples 1 and 2. In the notation of Section 2, this task is represented by the following parameters:

$$\langle work_O, span_O, work_N, span_N, D \rangle = \langle 900, 600, 120, 40, 690 \rangle.$$

It is to be scheduled upon $m = 10$ processors. In Example 4 we will show that the algorithm of Section 4 assigns the parameter m_N the value 3, and the parameter \mathcal{S}_N , a value $66\frac{2}{3}$. Hence our run-time scheduler starts out scheduling this task on 3 processors. If the task

behaves as specified by its $work_N, span_N$ parameters, then by Expression 1 the makespan is no more than

$$\frac{120 - 40}{3} + 40 = 26\frac{2}{3} + 40 = 66\frac{2}{3}$$

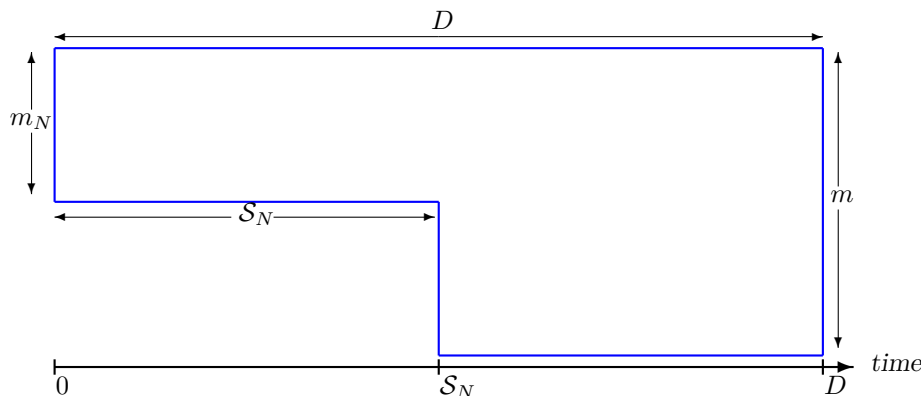
and hence the additional seven processors are not needed. If it does not complete by time-instant $66\frac{2}{3}$, all ten processors become available for this task to execute upon, and results in Section 4 allow us to conclude that the task does execute correctly, completing by the specified deadline at time-instant 690. ◀

3 Related Work

The approach to the modeling and run-time scheduling of parallelizable tasks that we are proposing here draws inspiration from research in the areas of *parallel computing*, *mixed-criticality scheduling*, and *probabilistic WCET*. As stated in Section 2 above, the problem of scheduling DAGs to minimize makespan (the P|prec| \mathcal{S}_{\max} problem in 3-field notation [15]) has been very widely studied in “traditional” scheduling theory. Given the inherent intractability of this problem [16] and the existence of a good approximation (as represented by List Scheduling [17] with its associated makespan bound – Inequality 1), the parallel computing community soon began to focus upon the work and span parameters as reasonable proxies for parallelizable computational workloads; this is one of the fundamental ideas that underpins our proposed approach.

The concept of specifying multiple values, which are considered trustworthy to different levels of assurance, to a task’s parameters was proposed by Vestal [10] and forms the basis of mixed-criticality scheduling theory. There is a large body of research exploring the Vestal model – see [14] for a survey. In studying the scheduling of mixed-criticality parallel tasks, Li et al. [23] first proposed a model in which each task is characterized by different work and span parameters at low and high criticality levels – it is this model that we are studying in depth here. (The overall context of the research in [23] is quite different from ours: while we are, in the terminology of [23], considering the scheduling of a single parallelizable real-time task with the objective of minimizing the number of processors used in the nominal case while concurrently guaranteeing to meet deadlines in the overloaded case, [23] was concerned with devising mixed-criticality scheduling algorithms with good *capacity augmentation bounds*.)

Our approach also draws upon ideas from the considerable body of prior research (e.g., [20, 21, 22]) on measurement-based techniques for estimating probabilistic worst-case execution time distributions (pWCET). The correctness of our scheduling framework very strongly depends upon the validity and accuracy of pWCET-estimation techniques, since we are in effect guaranteeing correct timing behavior (meeting deadlines) under the assumption that the more conservative estimations – $work_O$ and $span_O$ – are correct upper bounds. In contrast, incorrect estimations of $work_N$ and $span_N$ do not compromise correctness, although they could have an adverse impact on efficiency. Rather than being considered as estimations of worst-case parameter values, these parameters are perhaps closer in spirit to what Chisholm et al [25] have called *provisioned* parameter values and Li et al. [23], *nominal* parameter values – values that represent typical or common-case behavior and may be obtained by, e.g., somewhat inflating average-case parameter values.



■ **Figure 1** The parallel task begins execution at time-instant 0 with a deadline at time-instant D . It executes upon m_N processors over the interval $[0, \mathcal{S}_N)$, and upon m processors over the interval $[\mathcal{S}_N, D)$. (The x -axis thus denotes time, and the y -axis, the processors.)

4 Scheduling Algorithm Derivation and Analysis

Given a task characterized, as described in Section 2.1, by the parameters

$$\langle work_O, span_O, work_N, span_N, D \rangle$$

and m processors upon which to execute it, we discuss in this section how we should compute values of m_N and \mathcal{S}_N in order to ensure that the run-time scheduling algorithm described in Section 2.2 above completes execution of the task within D time units of its arrival. We will start out in Section 4.1 assuming that values for m_N and \mathcal{S}_N are already known, and derive sufficient conditions for ensuring timing correctness given these values of m_N and \mathcal{S}_N . We will then describe, in Sections 4.2, how values may be assigned to m_N and \mathcal{S}_N in a manner that ensures that these sufficient conditions are satisfied.

4.1 Sufficient Schedulability Conditions

Suppose that we are given values of m_N and \mathcal{S}_N (with $0 < m_N \leq m$ and $0 \leq \mathcal{S}_N \leq D$), and the run-time algorithm schedules the task on m_N processors using list scheduling. If the task completes execution within \mathcal{S}_N time units, correctness is preserved since $\mathcal{S}_N \leq D$. It remains to determine sufficient conditions for correctness when the task does not complete by time-instant \mathcal{S}_N ; this we do in the remainder of this section.

Figure 1 depicts the processors that are available for this task if it does *not* complete execution within \mathcal{S}_N time units, thereby resulting in the run-time scheduler awakening the $(m - m_N)$ processors that had been in sleep mode over $[0, \mathcal{S}_N)$. We will now derive conditions for ensuring that the task completes execution by its deadline at time-instant D when executing upon these available processors, given that its work parameter may be as large as $work_O$ and its span parameter, $span_O$.

Let $work'$ and $span'$ denote the work and span parameters of the amount of computation of the parallel task that remains at time-instant \mathcal{S}_N (these are > 0 , since the task is assumed to not have completed execution by time-instant \mathcal{S}_N). This remaining computation executes upon m processors; By Expression 1 the overall makespan is therefore bounded from above by

$$\mathcal{S}_N + \left(\frac{work' - span'}{m} + span' \right) \quad (3)$$

Since the remaining span at time-instant \mathcal{S}_N is $span'$, an amount $(span_O - span')$ of the critical path of the task has executed during $[0, \mathcal{S}_N]$. At each instant when the critical path is not executing, it must be the case that all m_N processors are busy executing tasks not on the critical path. Hence the total amount of execution occurring over $[0, \mathcal{S}_N]$ is at least

$$\left(\mathcal{S}_N - (span_O - span')\right) \times m_N + (span_O - span'),$$

from which it follows that

$$\begin{aligned} work' &\leq work_O - \mathcal{S}_N \times m_N + (span_O - span') \times m_N - (span_O - span') \\ &= work_O - \mathcal{S}_N \times m_N + (span_O - span') \times (m_N - 1) \\ &= work_O - \mathcal{S}_N \times m_N + span_O \times (m_N - 1) - span' \times (m_N - 1) \end{aligned} \quad (4)$$

Substituting Inequality 4 into the Expression 3, we obtain the following upper bound on the overall makespan:

$$\begin{aligned} &\mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N + span_O \times (m_N - 1) - span' \times (m_N - 1) - span'}{m} + span'\right) \\ &= \mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N + span_O \times (m_N - 1) - span' \times m_N}{m} + span'\right) \\ &= \mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N + span_O \times (m_N - 1)}{m} - span' \times \frac{m_N}{m} + span'\right) \\ &= \mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N + span_O \times (m_N - 1)}{m} + span' \times \left(1 - \frac{m_N}{m}\right)\right) \end{aligned} \quad (5)$$

Since $m_N \leq m$, Expression 5 is maximized when $span'$ is large as possible; i.e., $span' = span_O$ (the physical interpretation is that the worst case occurs when no job on the critical path is executed prior to time-instant \mathcal{S}_N : instead the entire critical path executes after \mathcal{S}_N). Substituting $span' \leftarrow span_O$ into Expression 5, we get the following upper bound on the overall makespan:

$$\begin{aligned} &\mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N + span_O \times (m_N - 1)}{m} + span_O \times \left(1 - \frac{m_N}{m}\right)\right) \\ &= \mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N - span_O}{m} + span_O\right) \end{aligned}$$

Correctness is guaranteed by having this upper bound on the makespan be $\leq D$:

$$\begin{aligned} &\left(\mathcal{S}_N + \left(\frac{work_O - \mathcal{S}_N \times m_N - span_O}{m} + span_O\right)\right) \leq D \\ &\Leftrightarrow \left(\mathcal{S}_N - \frac{\mathcal{S}_N \times m_N}{m}\right) \leq \left(D - \frac{work_O - span_O}{m} - span_O\right) \\ &\Leftrightarrow \mathcal{S}_N \left(1 - \frac{m_N}{m}\right) \leq \left(D - \frac{work_O - span_O}{m} - span_O\right) \end{aligned} \quad (6)$$

Expression 6 above is thus the sufficient schedulability condition we seek: values of m_N and \mathcal{S}_N satisfying Expression 6 guarantee timing correctness.

4.2 Computing m_N and \mathcal{S}_N

We saw in Section 4.1 above that in order to ensure correctness, our scheduling algorithm should choose the parameters m_N and \mathcal{S}_N such that Condition 6 above is satisfied. Recall that an additional goal is *efficiency*: the smaller the value of m_N , the better, since the remaining $(m - m_N)$ processors can be placed in sleep mode. In this section we describe how our algorithm computes such a value.

One reasonable approach for assigning a value to the m_N parameter is by using the task's *nominal* work and span parameters $work_N$ and $span_N$. Assuming that these parameters bound the work and span values of a “typical” invocation of the task, it is guaranteed by Inequality 1 that upon m_N processors a typical invocation will have a makespan no greater than $((work_N - span_N)/m_N + span_N)$. We may hence assign \mathcal{S}_N a value as follows:

$$\mathcal{S}_N \leftarrow \left(\frac{work_N - span_N}{m_N} + span_N \right) \quad (7)$$

Substituting this value for \mathcal{S}_N into Expression 6, we get

$$\left(\frac{work_N - span_N}{m_N} + span_N \right) \times \left(1 - \frac{m_N}{m} \right) \leq \left(D - \frac{work_O - span_O}{m} - span_O \right) \quad (8)$$

as a sufficient schedulability condition. Since every term other than m_N is a constant in this expression, the expression can be algebraically simplified to a form that is a quadratic expression in m_N ; solving this quadratic expression, and taking the ceiling (since the number of processors m_N must be integral) yields the desired value. Once m_N is so computed, the value of \mathcal{S}_N may be obtained from Expression 7. We illustrate via an example; the algorithm for computing m_N and \mathcal{S}_N is provided in pseudo-code form after the example.

► **Example 4.** Consider once again the instance discussed in Examples 1 and 2:

$$\langle work_O, span_O, work_N, span_N, D \rangle = \langle 900, 600, 120, 40, 690 \rangle$$

to be scheduled upon $m = 10$ processors.

Substituting these values into Expression 8, we get

$$\begin{aligned} & \left(\frac{work_N - span_N}{m_N} + span_N \right) \times \left(1 - \frac{m_N}{m} \right) \leq \left(D - \frac{work_O - span_O}{m} - span_O \right) \\ & \equiv \left(\frac{120 - 40}{m_N} + 40 \right) \times \left(1 - \frac{m_N}{10} \right) \leq \left(690 - \frac{900 - 600}{10} - 600 \right) \\ & \equiv \left(\frac{80}{m_N} + 40 \right) \times \left(1 - \frac{m_N}{10} \right) \leq 60 \\ & \equiv 40 \cdot \left(\frac{2}{m_N} + 1 \right) \times \left(1 - \frac{m_N}{10} \right) \leq 60 \\ & \equiv 2 \cdot \left(\frac{2 + m_N}{m_N} \right) \times \left(\frac{10 - m_N}{10} \right) \leq 3 \\ & \equiv (2 + m_N) \times (10 - m_N) \leq 15m_N \\ & \equiv 20 + 8m_N - m_N^2 \leq 15m_N \\ & \equiv m_N^2 + 7m_N - 20 \geq 0 \end{aligned}$$

from which we obtain

$$m_N \geq \frac{-7 + \sqrt{129}}{2} \approx 2.18$$

Since the number of processors must be integral, we conclude that $m_N \leftarrow 3$. The corresponding value for \mathcal{S}_N is equal to

$$\left(\frac{work_N - span_N}{m_N} + span_N \right) = \left(\frac{120 - 40}{3} + 40 \right) = 26\frac{2}{3} + 40 = 66\frac{2}{3}$$

Algorithm 1: Computing values for m_N, \mathcal{S}_N .

Input: $(\langle work_O, span_O, work_N, span_N, D \rangle, m)$
Output: *failure*, or values for m_N, \mathcal{S}_N

```

1 begin
2   if  $(m < \lceil (work_O - span_O) / (D - span_O) \rceil)$  then
3     return (failure) /* The test of Inequality 1 cannot guarantee that
4       the deadline will be met on  $m$  processors */
5   end
6    $A \leftarrow span_N$ 
7    $B \leftarrow m \times (D - (span_O + span_N)) - (work_O - span_O) + (work_N - span_N)$ 
8    $C \leftarrow (-1) \times m \times (work_N - span_N)$ 
9    $m_N \leftarrow \lceil (-1 \times B + (\sqrt{B^2 - 4 \times A \times C}) / (2 \times A)) \rceil$ 
10   $\mathcal{S}_N \leftarrow span_N + (work_N - span_N) / m_N$ 
11  return( $m_N, \mathcal{S}_N$ )

```

Pseudo-code representation. It may be verified that Expression 8 can be rewritten to be of the form

$$A \times m_N^2 + B \times m_N + C \geq 0$$

with A, B , and C assigned the following values:

$$\begin{aligned}
 A &\leftarrow span_N \\
 B &\leftarrow \left(m(D - (span_O + span_N)) - (work_O - span_O) + (work_N - span_N) \right) \\
 C &\leftarrow -1 \times m \times (work_N - span_N)
 \end{aligned}$$

The pseudo-code in Algorithm listing 1 finds the positive root of this quadratic inequality; the ceiling of which denotes the number m_N of processors needed – this computation occurs in Line 8. In Line 9 the value computed for m_N is used to determine the value to be assigned to \mathcal{S}_N .

Run-time complexity. Algorithm 1 comprises straight-line code with no loops or recursive calls. Hence given as input the parameters specifying a task, it is evident that Algorithm 1 has constant – $\Theta(1)$ – run-time.

5 Achieving Greater Efficiency: A More Aggressive Approach

In an attempt to achieve efficiency (reducing the number of processors used in the “common case”) while maintaining correctness (guaranteeing to meet deadlines provided task behavior does not exceed the worst-case bounds of $work_O$ and $span_O$), the approach derived in Section 4.2 above uses the nominal parameter values $work_N$ and $span_N$ to assign values to m_N and \mathcal{S}_N . In this section, we propose a more aggressive approach to achieving perhaps greater efficiency without compromising correctness in any manner. This more aggressive approach is based upon exploiting insights regarding (i) the probabilistic characterization of the run-time behavior of the system; and (ii) the typical behavior of List Scheduling.

The probabilistic characterization of run-time behavior. As discussed in Section 2 (and illustrated in Examples 1 and 2), we may have a *probability* associated with the likelihood that the $work_N$ and $span_N$ parameter values are correct. We may, for example be able to assert that there is a ≤ 0.05 probability that the actual work will exceed $work_N$, and a ≤ 0.05 probability that the actual span will exceed $span_N$. Now this threshold probability of 0.05 may have been selected because we desire that the probability that the $(m - m_N)$ sleeping processors will need to be awakened be ≤ 0.05 . If so, the method for computing \mathcal{S}_N and m_N described in Section 4.2 above may be overly conservative since the work and span distributions may not be perfectly correlated – if they are not, the probability that both the work would exceed $work_N$ **and** the span exceed $span_N$, during a particular execution of the task is smaller than 0.05. (In the extreme if the two distributions are more or less independent, the probability is closer to 0.05^2 which equals .0025, a value that is far smaller than the sought-for threshold probability of 0.05.)

Some observations on List Scheduling. Assuming that the actual work and span parameters of the computation do not exceed $work_N$ and $span_N$ respectively, in Section 4.2 we used Expression 1 to assign values to m_N and \mathcal{S}_N in a manner guaranteeing that the computation will complete execution within an interval of duration \mathcal{S}_N upon m_N processors. Note that Expression 1 is an *upper bound* on the makespan of a List Scheduling generated schedule of a DAG; this upper bound is tight only for DAGs possessing a very specific structural form and/ or List Scheduling making a particular sequence of scheduling decisions (and then only if each node of the DAG executes for its entire WCET). Simulation experiments using randomly-generated graphs seem to indicate that these structures and scheduling decisions are relatively rare; for randomly-generated graphs, the makespans of actual list-scheduling generated schedules tend to cluster closer towards the lower end of the interval between the upper bound of Expression 1 and the obvious lower bound of

$$\max\left(\frac{work}{m}, span\right), \quad (9)$$

even if each node of the DAG does actually execute for its entire WCET. To illustrate this, we randomly generated 1000-node DAGs with varying numbers of edges in the manner described in Section 5.1 below; for each, we computed the lower bound of Expression 9, the actual makespan using a list scheduling implementation, and the upper bound of Expression 1, for scheduling the DAG upon a 10-processor platform. The results are listed in Table 1. The right-most column – the one titled “Ratio” – denotes the fraction of the interval between the lower bound and the upper bound upon which the actual makespan encroaches.

More aggressive computation of m_N and \mathcal{S}_N . We highlight the fact that being too optimistic in assigning values to \mathcal{S}_N and m_N does *not* compromise correctness: the sole effect is upon efficiency in terms of the number of processors we are able to maintain in sleep mode, and the likelihood that these processors will need to be switched on during some run of the system. Hence one possible –more aggressive– approach towards achieving greater resource efficiency during run-time would be to assign \mathcal{S}_N a value between the lower and upper bounds of Expressions 9 and 1 as follows (rather than according to Expression 7):

$$\mathcal{S}_N \leftarrow \max\left(\frac{work_N}{m_N}, span_N\right) + \alpha \cdot \left[\left(\frac{work_N - span_N}{m} + span_N\right) - \max\left(\frac{work_N}{m_N}, span_N\right)\right] \quad (10)$$

with $\alpha, 0 \leq \alpha \leq 1$ a “tuning” parameter: the smaller the value of α , the more aggressive the choice of \mathcal{S}_N . (An intuitive interpretation of the tradeoff here is that the smaller the

■ **Table 1** Actual makespan, and lower and upper bounds, of randomly-generated DAGs upon a 10-processor platform. Each graph has 1000 vertices; each row corresponds to 1000-vertex DAGs with the number of edges specified in the first column. The last column denotes the ratio (actual makespan - lower bound) \div (upper bound - lower bound) – small values denote that the actual makespan is close to the lower bound.

(The experiments used to generate this table are detailed in Section 5.1.)

# edges	M A K E S P A N			Ratio
	Lower (Exp. 9)	Actual	Upper (Exp. 1)	
977	2627	2667	2818	0.208
2017	2539	2587	2889	0.137
4921	2567	2603	3222	0.055
9935	2554	2709	3725	0.132
20094	2599	2977	4774	0.174
39935	4056	4113	6154	0.027
50036	4454	4480	6491	0.013
60212	5674	5674	7658	0.000

value of α , the greater the number of processors we can switch off, but the greater the likelihood that they will need to be awakened during some execution of the task.) For the randomly-generated DAGs of Table 1, a value of $\alpha \geq 0.208$ would have been safe: during run-time the sleeping processors are not awakened as long as the task’s run-time behavior does not violate its nominal parameters $work_N$ and $span_N$.

If \mathcal{S}_N is assigned a value according to Expression 10 rather than Expression 7, it is no longer the case that solving Expression 8 yields the desired value of m_N . We have not attempted to derive a closed-form solution for m_N when Expression 10 is used in place of Expression 7; rather, we iterate through candidate values for m_N over the range $[1, m]$, stopping at the first such value for which this value for m_N , and the resulting value for \mathcal{S}_N computed according to Expression 10, causes Condition 6 to evaluate to true. This more aggressive approach to computing m_N and \mathcal{S}_N therefore has run-time complexity $\Theta(m)$ where m denotes the number of processors available; a straightforward application of the idea of binary search reduces this to $\Theta(\log m)$.

5.1 The Experiments Reported in Table 1

We now briefly describe the experimental procedure used to generate the data populating Table 1. Graphs were synthesized using a DAG-generating variant of the well-known Erdős-Rényi method [26] for generating random graphs. The Erdős-Rényi method, given parameters (n, p) , yields a graph on n vertices in which each edge has an independent probability p of existence. We modified this method to generate *directed acyclic* graphs with a target number of edges. Specifically,

- The number of vertices in the DAG, n , the maximum WCET parameter for a vertex w , and the desired number of edges e , are specified. The number of processors m upon which the DAG is to be scheduled is also specified.
- Each vertex is assigned a WCET parameter that is a randomly and uniformly drawn integer over the range $(1, w)$.

- The parameter p , denoting the probability of existence of each edge, is computed as follows:

$$p \leftarrow \frac{2e}{n \times (n - 1)} .$$

The idea is that since a DAG with n vertices has a maximum of $n/2 \times (n - 1)$ edges, if we were to create each such edge independently with probability p , on average the desired number e of edges would be created.

- All edges are assumed to be directed from the lower-indexed vertex to the higher-indexed vertex (hence a topological sorting of the DAG could yield the vertices in order of increasing index). Each such edge is created with probability p , as follows:

```

for i := 1 to n
  for j := (i+1) to n
    create edge (i,j) with probability p

```

- The work and span parameter of the generated DAG are computed, assuming that each vertex executes for exactly its WCET parameter value (i.e., WCET parameters are taken to represent the actual execution duration, rather than an upper bound on the execution duration). Using these values, a lower bound on makespan as given by Expression 9, and an upper bound as given by Expression 1, for the DAG upon the specified number m of processors are computed.
- A schedule of the DAG upon m processors using a standard implementation of list scheduling is generated (once again assuming that each vertex executes for exactly its WCET parameter value). The makespan of the resulting schedule is recorded.
- Each data-point reported in Table 1 was obtained by generating one hundred such graphs, computing the reported parameters upon each, and taking their averages.

6 Summary and Conclusions

Although DAG-based models for representing parallelizable real-time code have proved very popular in the real-time scheduling theory community, they suffer from several shortcomings that restrict their usefulness in representing some kinds of real-time code. In this paper, we have explored an alternative model, one that is based upon characterizing a task by just two parameters – work and span – with two estimates on upper bounds on the value of each parameter – one that may be very large but is trust-worthy to a very high level of assurance, and a second that is smaller and is more reflective of typical or nominal behavior. We have developed an algorithm for scheduling tasks that are so modeled upon a dedicated cluster of processors in a manner guaranteeing *correctness* – deadlines are always met provided run-time behavior does not violate the high-assurance bounds – while striving for *efficiency* – many processors can remain in sleep mode much of the time, only being switched on in rare circumstances when run-time behavior exceeds the normal bounds.

The model for representing parallelizable code that is being proposed in this paper, and the associated run-time scheduling algorithm, is particularly suitable for a certain kind of real-time application: one that repeatedly (i.e., periodically or sporadically) monitors the external environment seeking to detect some particular kind of anomalous sensory input. Most of the time the sought-for input is not detected, and not much computation needs to be performed. But on the rare occasions when the anomalous input is detected, considerable additional processing of such input is necessary; furthermore, such processing is highly parallel in nature. Example applications of this kind include real-time intrusion detection,

vision-based monitoring systems, etc. For such systems, we would expect that the nominal workload, as represented by the $work_N$ and $span_N$ parameters, is quite small and may not exhibit much parallelism; however the workload upon “overload” (i.e., when the monitored-for condition occurs) is quite intensive ($work_O$ is large) but exhibits considerable parallelism (i.e., $span_O$ is relatively small compared to $work_O$: equivalently, the ratio $work_O/span_O$ is large). If the system is hard-real-time, resource allocation for guaranteeing correctness must be made under worst-case assumptions – the $work_O$ and $span_O$ parameters. However, much of these allocated resources will remain unused much of the time during run-time; by being able to determine in a timely manner precisely when these unused resources will be needed during run-time, our approach allows us to place these resources in sleep mode until needed.

We believe our main contribution here is the model for parallel tasks – the run-time scheduler is presented as proof-of-concept evidence of the potential benefits, in terms of resource-efficiency, of adopting this model. As future work we plan to demonstrate the model’s applicability in a wider range of settings: under different scheduling paradigms (such as global EDF and global Fixed-Priority). We are also working on further extending the task model if additional profiling data of the task’s run-time behavior is available (and known to be reliable). For example, straight-forward generalizations allow us to specify multiple sets of parameter values at different probability thresholds (rather than just two sets of values) – are we able to develop scheduling strategies that can meaningfully exploit such additional information?

References

- 1 Karthik Lakshmanan, Shinpei Kato, and Ragnathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS*, pages 259–268. IEEE Computer Society, 2010.
- 2 Bjorn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2012.
- 3 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leem Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS 2012, pages 63–72, San Juan, Puerto Rico, 2012.
- 4 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer Publishing Company, Incorporated, 2015.
- 5 Jose Fonseca, Vincent Nelis, Gurulingesh Raravi, and Luis Miguel Pinho. A Multi-DAG model for real-time parallel applications with conditional execution. In *Proceedings of the ACM/ SIGAPP Symposium on Applied Computing (SAC)*, Salamanca, Spain, April 2015. ACM Press.
- 6 Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems*, ECRTS ’15, pages 222–231, Lund (Sweden), 2015. IEEE Computer Society Press.
- 7 Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems*, ECRTS ’15, pages 222–231, Lund (Sweden), 2015. IEEE Computer Society Press.
- 8 Robert I. Davis, Alan Burns, and David Griffin. On the meaning of pWCET distributions and their use in schedulability analysis. In *Proceedings 2017 Real-Time Scheduling Open Problems Seminar (RTSOPS)*, 2017.

- 9 Alan Burns and Robert Davis. Mixed-criticality systems: A review (9th edition). <http://www-users.cs.york.ac.uk/~burns/review.pdf> (Accessed on Aug 29, 2017), 2017.
- 10 Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the Real-Time Systems Symposium*, pages 239–243, Tucson, AZ, December 2007. IEEE Computer Society Press.
- 11 Sanjoy Baruah. The federated scheduling of systems of conditional sporadic dag tasks. In *Proceedings of the 15th International Conference on Embedded Software (EMSOFT)*, Amsterdam, the Netherlands, 2015.
- 12 James Anderson, Sanjoy Baruah, and Bjoern Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, San Francisco, CA, April 2009.
- 13 Alan Burns and Sanjoy Baruah. Towards a more practical model for mixed criticality systems. In *Proceedings of the International Workshop on Mixed Criticality Systems (WMC)*, December 2014.
- 14 Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, November 2017.
- 15 R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Mathematics*, 5:287–326, 1979.
- 16 J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.
- 17 R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.
- 18 J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.
- 19 Ola Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the 42nd ACM symposium on Theory of computing, STOC '10*, pages 745–754, New York, NY, USA, 2010. ACM.
- 20 S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *2001 IEEE Real-Time Systems Symposium (RTSS)*, pages 215–224, Dec 2001.
- 21 G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *2002 IEEE Real-Time Systems Symposium (RTSS)*, pages 279–288, 2002.
- 22 G. Bernat, A. Colin, and S. Petters. pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, The University of York, England, 2003.
- 23 Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.
- 24 Jing Li, Abusayeed Saifullah, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Analysis of federated and global scheduling for parallel real-time tasks. In *Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems, ECRTS '14*, Madrid (Spain), 2014. IEEE Computer Society Press.
- 25 M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2015 IEEE*, pages 305–316, Dec 2015.
- 26 P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290, 1959.