SOS: Introducing Shared Overrun Servers to Improve Probabilistic Scheduling

Sims Hill Osborne
Dept. of Computer Science, University of North Carolina, Chapel Hill, North Carolina, U.S.A.
shosborn@cs.unc.edu

***Abstract*—Probabilistic schedulers are needed in real-time scheduling when worst-case execution times (WCETs) cannot be determined, but such schedulers frequently face tractability issues or rely on the assumption that all tasks have independent execution times. To mitigate these problems, Shared Overrun Servers (SOSes) are introduced. SOSes execute portions of jobs that cannot be completed within a job's own budget, allowing jobs to share budget, even across cores. SOSes can be probabilistically analysed. Probabilistic properties and an example are given. The effects of complexities likely to arise in real systems are discussed.**

***Index Terms*—real-time systems, probabilistic scheduling**

I. Introduction

Traditional hard real-time schedulers guarantee that no deadline will ever be missed. Such guarantees rely on the assumption that worst-case execution times (WCETs) for individual tasks can be safely bounded. However, this assumption is unrealistic, particularly for multi-core systems. An alternative approach is to use probabilistic schedulers that model task costs as random variables and guarantee that deadline misses are improbable. The inspiration behind many probabilistic schemes is that one job having an unexpectedly long execution time will not always cause a deadline miss, and that if one job exceeding its *provisioned execution time* (PET) is rare, then multiple jobs doing so within a short time frame should be rarer still.

Unfortunately, analyzing how the runtime of every job can affect the scheduling of every other job can have combinatorial complexity. If probabilistic dependencies are considered—e.g. one job exceeding its PET makes it more likely that other jobs will do the same—the problem becomes even worse. With that in mind, we propose *Shared Overrun Servers* (SOSes) to address the question of how the runtime of one job can affect the scheduling of others without creating an intractable problem.

Shared Overrun Servers. SOSes schedule portions of jobs that fail to complete within the job's own PET; when a real-time job overruns its own budget, it can use the budget of an SOS job to continue executing. Like many well-known servers designed to schedule aperiodic work, such as the Constant Utilization Server [6], SOSes are an abstraction between a scheduler and the work to be scheduled. Rather than scheduling work directly, servers are allocated processor time which can be used to accomplish work. Previous works have considered servers in a probabilistic context, but we believe we are the first to use servers to allow budget-sharing across tasks and cores. Our method has similarities to slack reclamation algorithms such as BACKSLASH [7], but adds probabilistic analysis.

In our method, a task system is partitioned into subsystems, each with a unique SOS. The ability to complete a job that overruns its own PET is dependent only on the parameters of the subsystem's SOS and the behavior of other jobs within the subsystem. SOSes are a compromise between the pessimistic view that any job that takes too long to execute will miss its deadline and the impracticality of considering every possible scheduling interaction.

One goal of the partitioning process is to have only tasks with statistically independent execution times share an SOS. We thereby can avoid the assumption that all tasks are independent; we require only that some tasks be independent. Here we only touch on the problem, but the search for an effective partitioning algorithm will be the centerpiece of future work.

Contribution and Organization. In this paper, we introduce SOSes. In doing so, we lay foundations for addressing two challenges recently identified by Davis and Cucu-Grosjean [4]: developing methods for problems of a practical size and handling cross-task dependencies.

The remainder of this paper is organized as follows. In Sec. II, we define our task and platform model and provide an overview of probabilistic scheduling. In Sec. III, we outline the rules governing SOSes, our scheduling method, and give a simple example. In Sec. IV, we discuss complications that may arise in real systems. In Sec. V, we conclude and give directions for future work.

II. Background

In this section we give a brief overview of the motivations behind probabilistic scheduling and define our task and platform model.

Probabilistic scheduling motivation. Probabilistic schedulers are motivated by the difficulties of determining WCETs and the fact that even in many safety-critical

Work was supported by NSF grants CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, ONR grant N00014-20-1-2698, and funding from General Motors.

systems, guaranteeing that no task will ever miss a deadline may not be necessary. For multi-core architectures, there does not exist any means of WCET determination that is sound and scalable [9]. Only 34% of respondents to a 2020 survey on industrial real-time practices reported using static analysis—the only method capable of giving absolute guarantees—to estimate WCETs. In the same survey, 45% of respondents reported that even the most time-critical functions within their systems could miss some deadlines [1]. If schedules are built to guarantee no deadline misses even when some misses are acceptable, then some schedulable systems will be rejected as unschedulable.

Space restraints prevent us from highlighting individual works on probabilistic scheduling, but the recent survey by Davis and Cucu-Grosjean [4] gives a comprehensive summary of related work on probabilistic scheduling, including on the use of servers.

Task model. We consider a system τ consisting of n periodic tasks. Each task $\tau_i = (C_i, T_i, P_i)$ is defined by a random variable C_i for possible execution costs, its *period*, T_i, and an *acceptable deadline miss probability* P_i. The latter two parameters require only a brief explanation, given in the following paragraph. The cost parameter is more complex and is discussed further down.

Every task releases an unlimited number of *jobs*. Job releases of τ_i are separated by least T_i units of time and have a relative deadline of T_i. T_i and P_i together form a *probabilistic deadline* [4]. This deadline is the basis of the system's correctness condition.

Def. 1. A task system is *correct* if for all tasks τ_i, the probability of an arbitrary job failing to complete within T_i time units of its release is no greater than P_i. ◀

We assume that when a job does miss a deadline, it can be terminated safely and instantly. This assumption is not realistic, but it allows us to focus on the basic mechanics of SOS servers. We will revisit this assumption in the future.

Cost model. We consider tasks whose costs can be modeled via random variables. Sources of apparent randomness could include input variation, interference from other tasks, and even variations in platform temperature. We might have for τ_i

$$\Pr(C_i \leq 10) = 0.9 \text{ and}$$
$$\Pr(C_i \leq 20) = 0.99.$$

We express probabilities cumulatively to emphasize that what we know of C_i is not necessarily comprehensive. If being confident of τ_i's timing 99% of the time was not sufficient for our purposes, we could perform additional timing analysis work, allowing us to add

$$\Pr(C_i \leq 40) = 0.999.$$

to the distribution. If we can safely upper-bound τ_i's true WCET, there will be a point in the distribution with probability 1.0. For example,

$$\Pr(C_i \leq 80) = 1.0$$

would indicate a WCET no greater than 80. We assume, as do many probabilistic scheduling works, that cost variables are provided to us. For an overview of probabilistic timing analysis, we recommend survey papers by Cazorla et al. [3] and Davis and Cucu-Grosjan [5].

In addition to C_i, each task will be assigned a Provisioned Execution Time (PET), denoted $\mathcal{PET}_i$.[1] Note that while C_i describes an underlying task property, $\mathcal{PET}_i$ is a scheduling decision. PETs will be discussed further in Sec. III. Finally, each task has a *utilization* defined as

$$u_i = \frac{\mathcal{PET}_i}{T_i}.$$

Note that u_i, like $\mathcal{PET}_i$, is a scheduling decision rather than an inherent task property. Informally, utilization gives the long-term share of a processor that a task will need to execute, *assuming that it never overruns its PET.*

We require that tasks sharing an SOS, i.e. tasks in the same subsystem, have execution times such that PET overruns are independent, i.e the probability that a job of τ_i requires execution time greater than $\mathcal{PET}_i$ is not dependent on other jobs' PET overruns or lack thereof. How to determine PETs that allow this criterion to be met has been previously addressed by Liu et al. [8].

Scheduling model and hardware platform. In general, we will be using a partitioned scheduling model. In partitioned scheduling, each task is assigned to one core. Tasks are then scheduled on each core using a unicore scheduling algorithm. Specifically, we will be using partitioned earliest-deadline-first (P-EDF) scheduling. Under P-EDF, the system is schedulable with no missed deadlines assuming that no task overruns its PET so long as no core has utilization greater than 1.0. However, there will be times when tasks may migrate between cores. We will discuss scheduling rules further in Sec. III.

Our hardware platform π consists of m identical cores. We assume that cross-core effects that could influence task execution times are either included in execution costs or mitigated at the hardware level.

III. Scheduling with Shared Overrun Servers

In this section, we outline our proposed scheduling method, including the rules governing SOSes. The method given here is not complete, but is intended as a preview of future work. We prove some probabilistic properties that result from our method and discuss a very simple example.

[1]We use standard type to discuss PETs in general and the caligraphic form to denote a specific variable.

SOS definition. An SOS $S_A = (B_A, T_A, A)$ is defined by a *budget* B_A, a *period* T_A and a set of associated tasks A from τ. We require that for all $\tau_i \in A$, T_i be an integer multiple of T_A. While individual tasks are indexed by lower-case letters, SOSes are indexed with capitals. The *utilization* of S_A is denoted as

$$u_A = \frac{B_A}{T_A}.$$

SOS jobs are released periodically, but self-suspend so long as no job within A overruns its PET. When an overrun occurs, the SOS job becomes active and is scheduled using P-EDF like any other job, allowing the overrun job to continue executing so long as budget remains. Budget is consumed while the SOS job executes. At the SOS job's deadline, any remaining budget vanishes. SOSes provide a framework that will make it easier to produce probabilistic scheduling guarantees.

Scheduling method. Here we outline our scheduling method. While we have listed the steps below sequentially, they are in fact closely interrelated. Effective algorithms will need to consider the steps below holistically. **1)** Separate tasks into subsystems. **2)** Assign each task a PET. **3)** Define an SOS for each subsystem. **4)** Assign tasks and SOSes to cores. Tasks in the same subsystem do not need to be on the same core. For each core, total utilization of all tasks and SOSes must be no greater than 1.0.

Once these four steps have been completed, tasks and SOSes can be scheduled online using the P-EDF scheduling algorithm. PETs and SOS budgets are enforced by the operating system. If a job is not completed within its allocated time, then it is terminated. Limiting per-core utilization to 1.0 guarantees that if all jobs complete within their allocated time, scheduling in EDF order will guarantee no missed deadlines.

Deadline miss probabilities. We now show the deadline miss probabilities when SOSes are used in a simple case where τ has been partitioned so that A consists of two periodic tasks, τ_1 and τ_2, that share a common period and can be scheduled with no deadline misses so long as no PET is overrun. Common periods allow for a compact example; we will consider other cases in future work. We prioritize τ_1 over τ_2 for access to SOS jobs. If both tasks are assigned to a single core, prioritization can occur naturally, since one must execute before the other. If jobs are assigned to different cores, additional rules to determine priority may be necessary. With τ_1 prioritized over τ_2,[2] determining the probability the a τ_1 job will miss its deadline is trivial. We state it without proof.

Theorem 1. *Given periodic jobs τ_1 and τ_2 such that $T_1 = T_2$ where τ_2 has access to an SOS job only if the SOS job is not needed by τ_1, the probability of a job of τ_1 missing its deadline is given by*

$$\Pr(C_1 > \mathcal{PET}_1 + B_A)$$

[2]Cases where tasks have equal access to SOSes are left to future work.

The probability for τ_2 missing deadlines is slightly more complicated, as it depends on the timing behavior of the τ_1 job released at the same time. The following proof sketch relies on well-known probability axioms. We pessimistically assume, for the sake of brevity, that any overrun will completely consume an SOS job.

Theorem 2. *Given periodic jobs τ_1 and τ_2 such that $T_1 = T_2$ where τ_2 has access to an SOS job only if the SOS job is not needed by τ_1, the probability of a job of τ_2 missing a deadline is given by*

$$\Pr(C_2 > \mathcal{PET}_2 + B_A) + \Pr(C_1 > \mathcal{PET}_1) \\ \cdot [\Pr(C_2 > \mathcal{PET}_2) - \Pr(C_2 > \mathcal{PET}_2 + B_A)]$$

Proof sketch. A τ_2 job will be tardy if $C_2 > \mathcal{PET}_2 + B_A$ holds or jobs of both tasks require execution time greater than their respective PETs. The probability of this event is given by

$$\begin{aligned}
&\Pr\big(C_2 > \mathcal{PET}_2 + B_A \;\vee \\
&(C_2 > \mathcal{PET}_2 \;\wedge\; C_1 > \mathcal{PET}_1)\big). \\
&= \{\text{Since } \Pr(X \vee Y) = \Pr(X) + \Pr(Y) - \Pr(X \wedge Y).\} \\
&\Pr(C_2 > \mathcal{PET}_2 + B_A) \\
&+\Pr(C_2 > \mathcal{PET}_2 \;\wedge\; C_1 > \mathcal{PET}_1) \\
&-\Pr(C_2 > \mathcal{PET}_2 + B_A \;\wedge\; C_2 > \mathcal{PET}_2 \;\wedge\; C_1 > \mathcal{PET}_1) \\
&= \{\text{Since } C_2 > \mathcal{PET}_2 + B_A \rightarrow C_2 > \mathcal{PET}_2\}. \\
&\Pr(C_2 > \mathcal{PET}_2 + B_A) \\
&+\Pr(C_2 > \mathcal{PET}_2 \;\wedge\; C_1 > \mathcal{PET}_1) \\
&-\Pr(C_2 > \mathcal{PET}_2 + B_A \;\wedge\; C_1 > \mathcal{PET}_1) \\
&= \{\text{Since for independent events,} \\
&\quad \Pr(X \wedge Y) = \Pr(X) \cdot \Pr(Y).\} \\
&\Pr(C_2 > \mathcal{PET}_2 + B_A) \\
&+\Pr(C_2 > \mathcal{PET}_2) \cdot \Pr(C_1 > \mathcal{PET}_1) \\
&-\Pr(C_2 > \mathcal{PET}_2 + B_A) \cdot \Pr(C_1 > \mathcal{PET}_1) \\
&= \{\text{Using algebraic manipulation}\}. \\
&\Pr(C_2 > \mathcal{PET}_2 + B_A) + \Pr(C_1 > \mathcal{PET}_1) \\
&\cdot\; [\Pr(C_2 > \mathcal{PET}_2) - \Pr(C_2 > \mathcal{PET}_2 + B_A)]
\end{aligned}$$

□

Example. We now give a simple example to show the potential advantages of scheduling with SOSes. Let τ_1 be a periodic task such that $\tau_1 = (C_1, 40, 0.05)$ where C_1 has the cumulative distribution function

$$\Pr(C_1 \leq 10) = 0.9 \tag{1}$$

$$\Pr(C_1 \leq 20) = 0.99. \tag{2}$$

Let τ_2 be a second periodic task with identical parameters. Assume that both jobs can be safely dropped should they fail to complete. We need to define PETs and an SOS such that the probabilistic deadlines are respected.

One possibility is to define $\mathcal{PET}_1 = \mathcal{PET}_2 = 20$ and no SOS. Exps. 1 and 2 can be restated as

$$\Pr(C_1 > 10) = 0.1 \text{ and } \Pr(C_1 > 20) = 0.01.$$

It follows that all jobs would have a probability of 0.01 of missing deadlines and the two tasks would have a combined utilization of 1.0 and require full use of one processor.

A second option is to define

$$S_A = (10, 40, \{\tau_1, \tau_2\}) \text{ and } \mathcal{PET}_1 = \mathcal{PET}_2 = 10.$$

In this case, it follows from Theorems 1 and 2 that jobs of τ_1 have probability 0.01 of missing deadlines and jobs of τ_2 a 0.019 probability of missing deadlines, meeting the probabilistic deadlines for both tasks. However, the total utilization in this case is only 0.75, leaving some processor capacity available for other work.

While this example is simple, it illustrates the basic mechanics of SOSes and their ability to improve on the pessimistic view that a task overruning its PET must lead to a deadline miss. Next, we discuss challenges that will need to be addressed to make SOS servers a viable option for industrial systems.

IV. Complications

In this section, we discuss task system characteristics we plan to tackle in future work.

Defining task subsystems. How to divide τ into subsystems is a critical question, since all other scheduling decisions will follow from this step. We will need to consider independence of tasks, whether all tasks will be able to fit on a single processor, and whether all tasks within a subsystem share a common period. Subsystems with more than two tasks will need more complex rules governing the priority for access to the SOS in cases where multiple PET overruns occur.

Multiple cores. In the example of Sec. III, both tasks and the SOS could be scheduled on a single processor. However, a shorter period for τ_1 and τ_2 could make it impossible to schedule both tasks and the SOS on a single processor. The issue could be dealt with by placing τ_1 and S_A on one processor and τ_2 on a second processor. In that case, τ_2 would need to migrate to use S_A's budget, incurring an overhead cost. This overhead cost can be accounted for by increasing $\mathcal{PET}_2$ or B_A using techniques pioneered by Brandenburg [2].

Multiple periods and sporadic tasks. In our example, where two periodic tasks share a common period, determining when an SOS job can be used is simple; has it already been used within the same period? To some extent, this simplicity can be maintained in a real periodic system by defining tasks subsystems to contain only tasks with common periods. However, doing so will not be an option in systems that have few common periods. If we consider sporadic task systems, scheduling becomes still more complex. In both cases, more nuanced rules for when SOS jobs release and how SOS budgets are replenished will be necessary.

Jobs must complete. We have assumed that if a job cannot complete on time, it can be terminated safely and instantly. However, in a real system it may be necessary for a job to run to completion even after missing its deadline. We will consider how to deal with late jobs in future work.

Less pessimistic overruns. In Theorem 2 and our example, we assume that if a task overruns its budget, it will consume and entire SOS budget. This assumption is highly pessimistic. In future work, we will consider the case where overrunning jobs consume only a portion of an SOS job, leaving the remainder for other jobs.

V. Conclusion

In this paper, we have proposed Shared Overrun Servers as a means to allow tasks to share execution budgets and taken the first steps towards using them to improve probabilistic scheduling. In future works we will solidify this idea by giving a complete SOS ruleset, analyzing probabilistic deadlines in the presence of SOSes, and giving algorithms to determine task PETs and SOS parameters.

References

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. Davis. An empirical survey-based study into industry practice in real-time systems. In *RTSS 2020*. IEEE, 2020.

[2] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, UNC Chapel Hill, 2011.

[3] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1):14:1–14:35, February 2019.

[4] R. Davis and L. Cucu-Grosjean. A survey of probabilistic schedulability analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):04–1–04:53, 2019.

[5] R. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03–1–03:60, 2019.

[6] Z. Deng and JW-S Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium*, pages 308–319. IEEE, 1997.

[7] C. Lin and S. Brandt. Improving soft real-time performance through better slack reclaiming. 2005.

[8] R. Liu, A. F. Mills, and J. H. Anderson. Independence thresholds: Balancing tractability and practicality in soft real-time stochastic analysis. In *RTSS 2014*, pages 314–323, 2014.

[9] R. Wilhelm. Real time spent on real time (invited talk). In *RTSS 2020*. IEEE, 2020.