Proceedings of the

14th Junior Research Workshop on Real-Time Computing

JRWRTC 2021

https://rtns2021.univ-nantes.fr/jrwrtc/

April 8, 2021

Message from the workshop chairs

Welcome to the 14th Junior Researcher Workshop on Real-Time Computing, held in conjunction with the 29th International Conference on Real-Time and Network Systems (RTNS) in April 2021. The workshop provides an informal environment for junior researchers, where they can present their ongoing work in a relaxed forum and engage in enriching discussions with other members of the real-time systems community.

We would like to take this opportunity to express our gratitude to the members of the Program Committee listed below for thoroughly reviewing all the submitted papers. We would also like to thank all the authors who submitted their work to the workshop and hence contributed to its success. We wish you success in your scientific careers and we hope that the workshop will help you develop your ideas further. This year, JRWRTC accepted five peer-reviewed papers, which cover various topics of the real-time field such as scheduling, model checking, WCET analysis and vehicle networks.

Organizing this workshop would not have been possible without the help of many people. First, we would like to thank the Steering Committee of RTNS 2021, with special thanks to Liliana Cucu-Grosjean for her guidance and advice in the organization of the workshop. We would also like to thank the General Chair, Audrey Queudet, and the rest of the Local Organizing Committee, Sébastien Faucou, Jean-Luc Béchennec, Mikaël Briday, Anne-Marie Deplanche, and Houssam Eddine Zahaf, for all the time they put in coordinating the details and ensuring everything came together smoothly.

On behalf of the Program Committee, we wish you a pleasant workshop. May the environment be stimulating, with engaging presentations and fruitful discussions.

Catherine Nemitz, University of North Carolina at Chapel Hill, USA

Roberto Medina, Huawei Technologies, France

JRWRTC 2021 Workshop Chairs

Program Committee

Benjamin Rouxel – University of Amsterdam, Netherlands

James Orr – Washington University in St. Louis, USA

Hai Nam Tran – Université de Bretagne Occidentale, France

Kuan-Hsun Chen – TU Dortmund, Germany

Daniel Casini – Scuola Superiore Sant'Anna, Italy

Lea Schönberger – TU Dortmund, Germany

Ashik Ahmed Bhuiyan – University of Central Florida, USA

Tanya Amert – UNC – Chapel Hill, USA

Tobias Blaß – Bosch Corporate Research, Germany

Anam Farrukh – Boston University, USA

Aaron Willcock – Wayne State University, USA

Mohamed Irfan Abdulla – E-Cobot, France

Pierre-Julien Chaine – Onera, France

Kevin Zagalo – INRIA, France

Matheus Ladeira – ENSMA, France

Frédéric Fort – Université de Lille, France

Contents

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*

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

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*

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

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

$$\Pr(C_1 \leq 10) = 0.9 \quad (1)$$

$$\Pr(C_1 \leq 20) = 0.99. \quad (2)$$

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

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.

Modeling Single Event Upsets in UPPAAL SMC for Real-time DAG Scheduling

Lukas Miedema
l.miedema@uva.nl

Benjamin Rouxel
b.rouxel@uva.nl

Clemens Grelck
c.grelck@uva.nl

Abstract

Real-time cyber-physical systems have become ubiquitous. As such systems are often safety-critical, designers must include mitigations against various types of hardware faults, including *Single Event Upsets* (SEU). SEUs are transient faults that only momentarily affect a single processor, after which the processor returns to normal operation. The effect of a SEU may manifest itself in the job running on that processor as incorrect output.

We present a new approach for analyzing schedulability using UPPAAL in *Stochastic Model Checking* (SMC) mode while including mitigations for SEUs using job restarts into the scheduler. By restarting a job after experiencing a SEU, the fault-free response of the job is delayed, potentially multiple times. We propose a method that informs the system designer about the distribution of the applications makespan, including the probability that a given deadline will be met in the presence of SEUs.

Keywords

single upset event, soft error, transient fault tolerance, UPPAAL, timed automaton, DAG scheduling

1 Introduction

In this work we focus on online scheduling while dealing with *Single Upset Events* (SEUs) on symmetric multi-core platforms. SEUs, also called *soft errors* or *transient faults*, are a type of fault which only momentarily affects a single processor and does not leave it in a degraded state. Such faults may, for example, be caused by cosmic rays [8]. The impact of a SEU is that the output of the running code may be erroneous, which can be disastrous for safety-critical systems.

In *Directed Acyclic Graph* (DAG) scheduling, the deadline is shared between a set of tasks. Our method does not allocate extra time per task to handle restarts, but instead requires moving the shared deadline to facilitate tasks restarting. It is up to the online scheduler to assign this extra time dynamically based on which tasks require a restart.

UPPAAL [1] is an existing tool for modeling, validating and verifying real-time systems, which has found use in the RTS community [7]. The tool can verify properties or give counterexamples, e.g. prove that a deadline will always be met. When mitigating SEUs, the number of restarts a task can require is not bounded. Instead, each number of restarts carries a probability. Our method does not require an upper bound on the extra time needed to facilitate restarts, unlike for example the Δ_f limit used by Mosse et al. [5]. Instead, our method provides insight into how likely it is that a given deadline will be missed without such an upper bound.

To estimate probabilities, we use UPPAAL in *Stochastic Model Checking* (SMC) mode. This mode is incompatible with models not designed to be used with SMC, like the UPPAAL RTS template library by Shan et al. [7]. Probability has been used before in real-time scheduling [2], and examples of this approach include using statistical means for determining a sensible upper bound estimate for the *Worst-Case Execution Time* (WCET) [3].

Contributions. We present a new method for representing a DAG of tasks and the hardware as an UPPAAL SMC model including SEUs, without putting limits on the number of job restarts. Our model includes an online scheduler implemented in the UPPAAL language, allowing the scheduler to react to the effects of job restarts caused by SEUs. For any given deadline, the UPPAAL SMC model can show the probability that the deadline will be met including SEU mitigations. It is then up to the application designer to choose an acceptable risk level.

2 Fault and Task Model

2.1 Task model and communication

The DAG of tasks has a single, constrained deadline $D \leq T$, allowing us to consider each execution of the DAG in isolation with exactly one job per task (without fault detection). Furthermore, for each task $\tau_i \in \tau$, a WCET value C_i must be available, which we use in our online scheduling algorithm. Our scheduler implements a global, fixed-priority, task-migrating and non-preemptive scheduling algorithm. The priority P_i of task τ_i is directly derived from the WCET value C_i. The higher the WCET, the higher the priority of that task. Let n be the number of tasks, and $i \in [0, n)$, we define the priority as $P_i = C_i \cdot n + i$, resulting in unique priorities due to the inclusion of i.

Communication between tasks is exclusively handled via the exchange of *tokens*, which may contain arbitrary data. Furthermore, we consider communi-

cation overhead negligible. In order to launch tasks multiple times, we assume all tasks to be stateless and free of side-effects.

2.2 Fault Detection and Mitigation

We assume that when a SEU occurs, it affects only one processor. While the SEU itself is transient, effects of the event may still be present and continue to affect the job. We assume that the SEU manifests itself as an incorrect result for that job (incorrect output tokens).

Let there be m processors, then each processor π_k where $k \in [0, m)$ has an associated SEU fault rate parameter λ_k. We model the inter-arrival times of faults with an exponential distribution as we are only concerned with SEUs and not with permanent degradation of the hardware. Exponential distributions are the only continuous memory-free distributions, hence knowledge about how long a processor has been fault-free does not impact the future probability of a fault in any way. Since we focus on symmetric multi-core systems, we treat the fault rate for each processor as identical ($\forall k \in [0, m) : \lambda_k = \lambda$).

We validate the output of a task using *Dual Modular Redundancy* (DMR) [6] by spawning two jobs for every task. If the content of the output tokens does not match, one or both jobs experienced a SEU while running. SEUs may also impact jobs in such a way that they never terminate. As such, jobs exceeding their WCET are also considered faulty and are terminated by a watchdog timer.

To recover from a fault, we use *Checkpoint-Restart*. The input tokens are always checkpointed and kept for the duration of the task. If a fault is detected, the produced tokens are discarded and the two jobs are started again using the original input tokens.

Our implementation of DMR runs the two jobs concurrently on separate processors, an approach also known as *Chip-level Redundant Multithreading* (CRT) [6]. As such, the use of DMR itself does not impact the *Worst-Case Response Time* (WCRT) of the task, assuming there are no SEUs requiring restarts.

3 Scheduling with Restarts

The number of restarts a particular job needs to endure before it is guaranteed to produce a result without a SEU is not bounded. The lack of such a bound makes the fault-free WCRT effectively infinite, and as such no longer a particular useful metric to work with. As a result, our technique only works with online scheduling.

3.1 Definitions

To simplify reasoning about DMR and checkpoint-restart, we introduce the fault detecting task $\overline{\tau}$, running on the fault detecting processor $\overline{\pi}$ as conceptual abstractions. The fault detecting task $\overline{\tau_i}$ is constructed from user-provided task τ_i, but can detect that a SEU has occurred. The $\overline{\tau_i}$ runs the user-provided task τ_i twice and compares the output tokens of both executions. Similarly, the $\overline{\pi_k}$ is a tuple consisting of two hardware processors. With these constructions in place, a fault detecting task can be scheduled on a fault detecting processor, hence we can reuse existing scheduling algorithms.

3.2 Schedulability Testing using Stochastic Model Checking

Our method informs the application designer about the probability of meeting the deadline for each execution of the DAG. To obtain this number, we model the entire application (including the scheduler) as a UPPAAL SMC model [1]. Tasks are treated as black boxes and are assumed to always need their entire WCET estimate. Furthermore, we do not model the individual processors or tasks, but instead we directly model the fault-detecting processors and fault-detecting tasks. We implement our global fixed-priority scheduler in the UPPAAL language, controlling transitions in the task automaton based on its position in the scheduler queue.

3.3 Representation as UPPAAL Model

Our method represents the DAG of tasks, the scheduler and the hardware as a composition of four UPPAAL templates:

1. a singleton scheduler;
2. for each fault detecting task $\overline{\tau_k}$, an instance of the `task` template;
3. for each fault detecting processor $\overline{\pi_f}$, an instance of the `processor` template;
4. for each dependency between two tasks, an instance of the `edge` template.

We show the task template in Figure 1. This template is parameterized with the task id i and the WCET estimate C_i of the task that it represents. Tasks start out in the `UnmetDependencies` state until all of their dependencies have been met. Then, they transition to the `Ready` state where they are available to be picked up by the scheduler and assigned to a processor to start running in the `Running` state. After some time, the task either finishes or transitions to a faulty state during execution (`RunningWFault` state), moving it back to the `Unscheduled` state for another try. This procedure sheds light on how faults are modeled: a SEU is triggered by the `processor` model (omitted here for brevity) after which the task running on that processor (if any) moves to the `RunningWFault` state by synchronizing on a channel. Synchronization forces other models in the system to transition as well when in a state with an outgoing edge waiting for the same channel. In this case, the `processor` model signals the `task_transient_fault[`i`]` channel, prompting the task $\overline{\tau_i}$ to transition to the `RunningWFault` state.

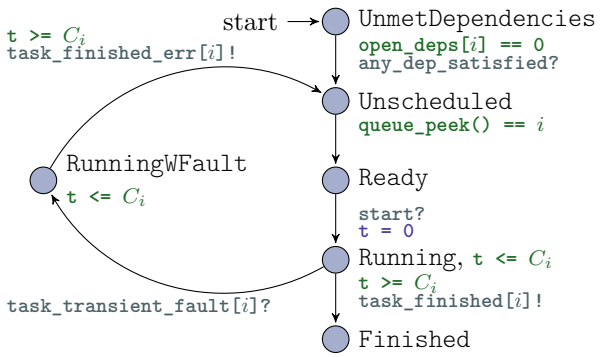


Figure 1: Summary of the UPPAAL model of a task execution. The task continues to run in an erroneous state ("RunningWFault") after experiencing a transient fault.

Running in this state does not change the task from the perspective of the scheduler. It is only when the task has completed that it either signals success ("`task_finished[`i`]!`") or error ("`task_finished_err[`i`]!`"), to which the scheduler can respond.

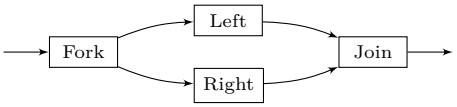
Task execution is represented as a *timed automaton*, where the state is composed of not just discrete variables but also of continuous (clock) variables. In the case of the task model this is the clock variable `t` for time. The clock `t` is set to 0 when transitioning to the `Running` state. The transitions out of the `Running` or `RunningWFault` state are guarded: $t \geq C_i$, enabling the transition only when the task has been running for its WCET estimate. A state invariant on the two running states of $t \leq C_i$ prevents lingering in the running states past the tasks WCET estimate.

Tasks may not always need their full WCET estimate to produce a result. However, modeling the early return of tasks would require knowledge about the frequency and distribution of such cases happening. Despite this, the WCET estimate provides a useful upper bound that lets us provide a lower bound probability that the deadline will be met.

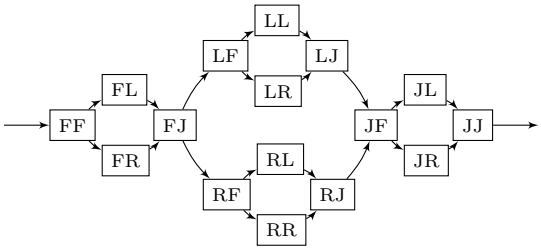
4 Evaluation

4.1 Experimental Setup

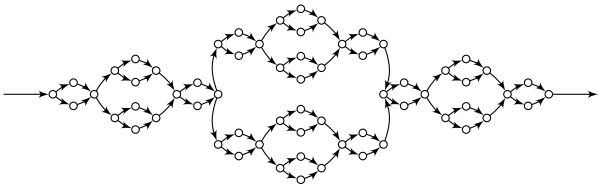
To test our technique we have created three synthetic test cases. As the potential gain by our technique scales with the number of tasks in the graph, we have tested three DAGs with varying critical path lengths. Each test case is a variation of the same fork-join graph with a different level of nesting, as shown in Figure 2. To focus just on the impact of restarting, we set the WCET estimate of each task to the same value of 1000 seconds. We do not give our DAGs a deadline or period, but are instead interested in the distribution of whole-DAG response times. Instead of obtaining a probability of meeting the deadline, we gain insight into what a sensible deadline might be together with the probability of meeting that deadline.



(a) Test case A: fork-join with four tasks



(b) Test case B: nested fork-join graph. Each of the tasks in the previous graph is replaced with the entire graph.



(c) Test case C: double nested fork-join graph. Each of the tasks in test case B is replaced with the entire graph of test case A.

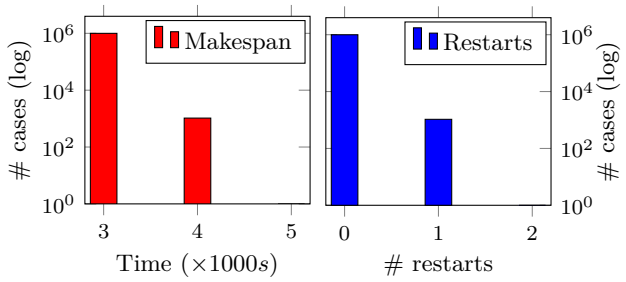
Figure 2: The various test cases, showing 4^1, 4^2 and 4^3 tasks

To execute the tests, we simulate a platform with 4 fault detecting processors (constructed from 8 hardware processors). Furthermore, we set the rate of transient faults for each processor to $\lambda = 10^{-3}/hour$, which is the highest permissible fault rate per RTCA/DO-178C for faults that have only minor criticality level [4].

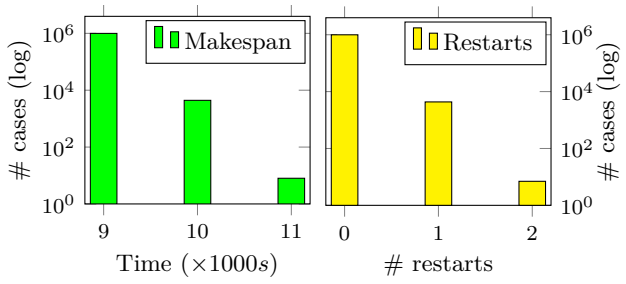
4.2 Results

Figure 3 shows the results of the UPPAAL SMC simulations. The distribution of the whole-DAG makespan (left) informs the application designer about the probability of meeting a particular deadline, or helps determining a suitable deadline. For example, let the deadline of test case C be $D_C = 3.0 \cdot 10^4 s$, the simulations can show us that the probability of missing this deadline is $1.45 \cdot 10^{-4}$. Or, if the deadline has not been determined yet, $D_B > 1.1 \cdot 10^4 s$ could be determined as a suitable deadline for case B. However, the exact bounds depend on the amount of risk acceptable in the domain of the application.

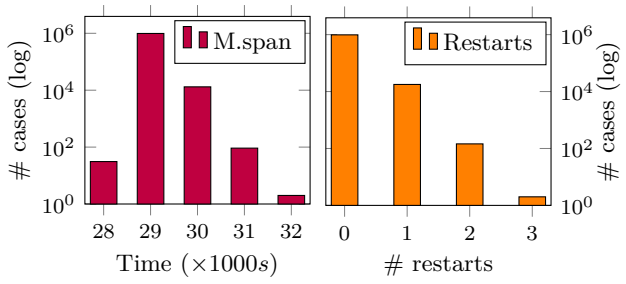
Our UPPAAL representation does not express any variance in the execution time of a task, which means that the whole-DAG execution time is always an integer multiple of a 1000 seconds. For all cases, the number of restarts has a very predictable impact on the makespan. Test case C (Figure 3c), however, also shows signs of something else: a number of runs manage to finish early. We explain this as a *timing anomaly* caused by some non-determinism in the



(a) Results for test case A



(b) Results for test case B



(c) Results for test case C

Figure 3: UPPAAL results for 10^6 SMC simulations. For each test case, the distribution of whole-DAG makespan is shown on the left. On the right, the number of restarts is shown.

completion order of tasks. When multiple tasks complete at the same time, the scheduler randomly handles one first and reassigns the processor due to a lack of clairvoyance. Even though the scheduler assigned the highest-priority task to the idle processor, it can only consider tasks that have all of their dependencies satisfied. Furthermore, our scheduler does not preempt, as such the release of a higher-priority task may must wait for a lower priority task to finish. Our fixed-priority online scheduling algorithm is not an optimal algorithm (without preemption), which allows this source of non-deterministic behavior to impact the whole-DAG execution time.

As the number of tasks in the DAG increases, so does the number of restarts. This makes sense: the longer the DAG runs, the higher the chance of a transient fault to occur.

5 Conclusion and Future Work

Transient faults in processors lend themselves well to be modeled using UPPAAL in SMC mode. We have shown in this paper how an existing SMC tool can be leveraged to get timing information of an application in the presence of SEUs.

Our approach models how the tasks are scheduled, including how an online scheduler may absorb the time lost due to a restart. However, for relatively small test cases (up to 64 tasks), Figure 3 shows a strong correlation between the number of restarts and the makespan. In future work, we will look at larger DAGs of tasks and investigate varying the amount of dependencies between the tasks. We suspect that, for more loosely connected graphs, SEUs can more often be solved by making use of idle processors. As such, for a given size of the DAG, we expect the number of restarts to no longer be a good predictor of the makespan.

Finally, our results are obscured by timing anomalies. In future work, we hope to mitigate these anomalies by implementing clairvoyance in our scheduler.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project).

References

[1] P. Bulychev et al. "UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata". In: *arXiv e-prints* (2012).

[2] R.I. Davis and L. Cucu-Grosjean. "A survey of probabilistic schedulability analysis techniques for real-time systems". In: *LITES: Leibniz Transactions on Embedded Systems* (2019).

[3] S. Edgar and A. Burns. "Statistical analysis of WCET for scheduling". In: *22nd IEEE RTSS*. 2001.

[4] A. Löfwenmark and S. Nadjm-Tehrani. "Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective". In: *Journal of Systems Architecture* 87 (2018).

[5] D. Mosse et al. "A nonpreemptive real-time scheduler with recovery from transient faults and its implementation". In: *IEEE Transactions on Software Engineering* (2003).

[6] Isil Oz and Sanem Arslan. "A Survey on Multithreading Alternatives for Soft Error Fault Tolerance". In: *ACM Comput. Surv.* (2019).

[7] L Shan et al. "RTLib: A Library of Timed Automata for Modeling Real-Time Systems". PhD thesis. Grenoble 1 UGA-Université Grenoble Alpe; INRIA Grenoble-Rhone-Alpes, 2016.

[8] F. Wang et al. "Single Event Upset: An Embedded Tutorial". In: *21st International Conference on VLSI Design*. 2008.

Toward a fine-grained execution model of real-time tasks

Zineb Boukili
zineb.boukili@univ-brest.fr
Lab-STICC, CNRS, UMR 6285,
Univ. Brest, Brest, France

Hai Nam Tran
hai-nam.tran@univ-brest.fr
Lab-STICC, CNRS, UMR 6285,
Univ. Brest, Brest, France

Alain Plantec
alain.plantec@univ-brest.fr
Lab-STICC, CNRS, UMR 6285,
Univ. Brest, Brest, France

ABSTRACT

Verification of real-time systems in scheduling analysis tools is generally based on the validation of timing constraints by the mean of scheduling simulation or feasibility test. Real-time tasks are modeled with their worst-case execution times and deadlines together with other properties depending on the choice of scheduler. Even though the current task model provides sufficient information to perform basic schedulability tests, it is inadequate to enforce a dynamic control to guarantee the normal operation of a system under hardware/software malfunctions or malicious cyber attacks. In this article, we present an approach to extend the current task model in the Cheddar scheduling analyzer with information of internal executions and data accesses. We demonstrate the applicability of our model by providing a case study as well as its realization in the architecture analysis and design language (AADL).

1 INTRODUCTION

Real-time critical systems (RTCS) are systems in which the correctness of their behavior depends not only on the logical results of computations but also on the physical time when these results are produced. They are qualified as critical because the absence of response is at least as serious as its incorrectness and the failure of such systems has unacceptable consequences for society. Typical examples of real-time systems include flight control systems, networked multimedia systems, command control systems, and real-time monitors.

Verification methods used for RTCS are mostly based on the early validation of timing properties. These methods are qualified as early verification because they are used during the design phase. However, this type of verification is insufficient to guarantee the proper functioning of a system during operation considering hardware/software malfunctions or malicious attacks.

Figure 1: Invalid behavior w.r.t data accesses

Figure 1 depicts such invalid behavior, on the left (a), a valid task, and on the right (b), the same task but with a delayed write data access. On the left side, the valid behavior of the task consists in two successive read accesses and one write access (a). An application malfunction or an intrusion attack can lead to an invalid behavior scenario as shown on the right side (b) where the writing operation has exceeded the expected time with a delta delay.

In our work, we focus on the detection of timing deviations at run-time by monitoring data accesses. Our approach bases on a fine-grained execution model of real-time tasks then check dynamically all data accesses according to declared timing constraints. This article presents the first step that extends the task model to take into account these elements in a scheduling analysis tool.

Problem statement: Real-time task models in scheduling analysis tools such as [6, 14, 16] do not allow the specification of data accesses and their related timing constraints. Scheduling analysis by the mean of simulation in these tools stops at the task level and does not give insights of internal task executions. Thus, scheduling analysis does not provide meaningful data for run-time monitoring as we expect to monitor data accesses and detect timing deviations.

Contribution: In this article, we present a real-time task model that takes into account data access by decomposing tasks into smaller units called *execution units*. We show how to extend the task model exists in the Cheddar scheduling analyzer with our proposals. In addition, we explain how to integrate our model in the architecture analysis and design language AADL[7]. Experiments are conducted with the research open-source avionics and control engineering (ROSACE) [12] case study and the OTAWA [2] static analyzer to show how data can be acquired for our model.

The rest of the article is organized as follows. Section 2 presents the background and positions our work. Section 3 presents our modeling approach. Section 4 explains our experimentation method and presents the results on the ROSACE case study. Finally, section 5 concludes the article and discusses future work.

2 BACKGROUND & RELATED WORK

In this section, we provide a brief summary of task models used in scheduling analysis tools. Then, we present an overview of the existing approaches focus on decomposing task execution into smaller units or phases.

2.1 Task model in scheduling analysis tools

The real-time task model presented in the seminal work of [10] has been widely implemented in real-time scheduling analysis tools such as [14], [6], [16], [4]. A task is characterized by two attributes: capacity and deadline. The capacity represents the worst-case execution time (WCET) of the task. The task must complete before a deadline, which can be either relative or absolute. Depending on the choice of scheduler, a task can have other attributes such as priority, period, and release time.

Several extensions of the basic task model have been implemented in the tools presented above to account for new elements such as cache-related preemption delay [15], [6], energy consumption [5], and security [1]. Nevertheless, the capacity remains the only attribute representing the execution of a task. In the next section, we present the approaches extending the execution model.

2.2 Modeling task execution

The idea of decomposing tasks executions into smaller units or phases has been proposed in [3, 11, 13]. In [13], the authors present the predictable execution model (PREM) [13] which co-schedules at a high level all active COTS (Commercial-Off-The-Shelf) components in the system, such as CPU cores, and I/O peripherals to permit predictable, system-wide execution. When a typical real-time task is executed on a COTS CPU, cache misses are unpredictable. This issue makes it difficult to avoid low-level contention for access to main memory. PREM aims to solve the problem of memory interference between peripherals and CPU tasks. The code for each task is divided into a set of N scheduling intervals classified into compatible and predictable intervals. First, each predictable interval is divided into two different phases. During the initial memory phase, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last-level cache. Second, the second phase is known as the execution phase. During this phase, the task performs useful computation without suffering any last-level cache misses.

Another way to model the execution of a task is the time interest points (TIP) [3] graph. The authors describe TIPsGraphs as an intermediate representation to transform the CFG (Control Flow Graph) representing the control flow of a task into a sequence of time intervals representing the timing aspects of the task execution. Typically, a TIP can be:

- Memory instructions (stores and loads), when the static analysis cannot guarantee that they will always result in AH.
- Memory instructions addressing shared variables, or data residing in a cache block that can be written by another task.
- Instructions for which the static analysis cannot guarantee that they will always result in a hit in the instruction cache.
- Pivot instructions.

In [11], the authors propose the acquisition execution restitution (AER) model. Tasks are divided into three distinct phases, namely acquisition (A) and restitution (R) which are communication phases (i.e., phases in which accesses to the memory are performed) and a single execution phase (E) which is a computation-only phase. The AER model is a generalization of the PREM model which considers a single core environment while the AER model executing in multi-core environments.

The discussed models are designed mainly to improve scheduling analysis and to reduce timing interference in real-time applications. But none of them provide additional information about data accesses that we want to monitor in our approach.

3 APPROACH

In this section, we explain our approach to decompose task executions into smaller units. Our model is implemented in the Cheddar scheduling analyzer, we propose also an implementation for the execution unit in AADL language using the behavioral annex. Cheddar [14] is an open-source real-time scheduling analysis tool which allows users to model software and hardware architectures of real-time systems, and to check their schedulability or other performance criteria.

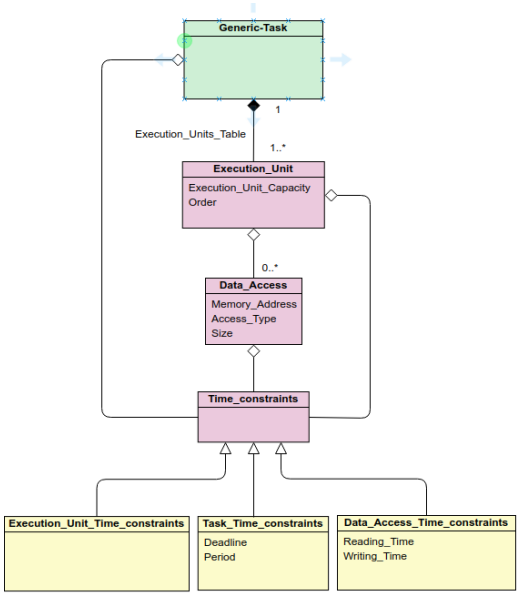


Figure 2: Suggested Task Representation Model

3.1 Cheddar Task Model Update

The original task model in Cheddar stops at the task level represented by the green box in Figure 2 and does not take into account the internal task design because the classic scheduling analysis does not require this information. In Cheddar, a *generic task* is modeled with the attributes allowing scheduling analysis on multi-processor platforms such as: capacity, priority, deadline and core mapping. Depending on the choice of scheduler, a task can have additional attributes including period, offset and release time. Interested readers can refer to the complete description of the hardware and software component supported by Cheddar in [8].

Our solution consists in representing task execution as a set of internal blocks called execution units. Each execution unit represents a portion of the task code. As depicted in Figure 2, each execution unit has a chronological order and a capacity (execution time). Within a period, a task includes a sequence of execution units, namely the compatible interval, the memory phase and the execution phase. Therefore an execution unit owns a Unit_type property that represents its particular phase.

Similarly, an execution unit can be composed of one or more data accesses, each one is characterized by a memory address of the accessed data, by the size of the data and by an access type, i.e. read or write access.

A task, an execution unit or a data access can all be associated with a time constraint specification. Thus, a time constraint is represented as an abstraction that can have three kind of concrete representations: Task_Time_Constraint specifies the Task time constraints with a deadline and a period. Data_access_Time_Constraint specifies data access time constraint with a reading or a writing time. Finally, Execution_Unit_Time_Constraint specifies execution unit kind of time constraint.

```
SCHEMA Execution_Units;
    TYPE Execution_Unit_Type = ENUMERATION
        OF (Compatible_phase,
            Memory_phase,
            Execute_phase,
            Non_identified);
    END_TYPE;
    ENTITY Execution_Unit
        SUBTYPE OF ( Named_Object );
        Unit_type : Execution_Unit_Type;
        Time_Constraint : Time_Constraint;
        Capacity : Natural;
        Order : Natural;
    END_ENTITY; ...
```

Figure 3: Execution Unit Entity

```
SCHEMA Data_Access;
    TYPE Data_Access_Type = ENUMERATION
        OF (Read_Data, Write_Data);
    END_TYPE;
    ENTITY Data_Access
        SUBTYPE OF ( Named_Object );
        Access_Type : Data_Access_Type;
        Memory_Address : Integer;
        Time_Constraint : Time_Constraint;
        Size : Integer;
    END_ENTITY; ...
```

Figure 4: Data Access Entity

To introduce these modifications into the Cheddar tool, we update its meta-model by adding the execution unit and data access entities as shown in the Figures 3 and 4[1].

3.2 Execution Unit Implementation in AADL

Our work includes the architecture analysis and design language (AADL) part, in which we implement the execution unit component using the behavioral annex. It provides a standard sub-language extension to allow behavior specifications to be attached to AADL components. The aim is to refine the implicit behavior specifications that are specified by the core of the language.

In AADL, the subprogram component may represent a portion of the source code, therefore we can implement our execution unit as a subprogram component. Then we describe the internal behavior of the execution unit as a state automaton with transitions, guards and actions as shown in the behavioral annex. To illustrate the idea, we consider an example of the execution unit with 2 data accesses (read and write). As shown, in Figure 5, and according to the behavior annex, the execution unit correspond to a subprogram, inside, we have two states each corresponds to data access operation

[1]These specifications are in EXPRESS (ISO 10303-11 [9]), the data modeling language that is used to implement the Cheddar meta-models

```
SUBPROGRAM Execution_Unit
END Execution_Unit;

SUBPROGRAM IMPLEMENTATION Execution_Unit.impl
annex Behavior_Specification {**
    states
        Read : state;
        Write: state;
    transitions
        Writing_operation:
        Read-[on dispatch]-> Write {
        computation(6ms); -- Access time to write data
        };
        Reading_operation:
        Write-[on dispatch]-> Read {
        computation(3ms); -- Access time to read data
        };
        **};
END Execution_Unit.impl
```

Figure 5: Execution Unit Implementation in AADL

that we may have in an execution unit, therefore the computation instruction is used to define the time constraint of the data access.

4 CASE STUDY: ROSACE

In this section, we describe our evaluation to measure the WCET of tasks and execution units. The evaluation is based on the ROSACE [12] case study using OTAWA [2] tool.

4.1 ROSACE Case Study

ROSACE [12] is a research open-source avionic control engineering case study, that goes from a baseline flight controller, developed in MATLAB /SIMULINK , to a multi-periodic controller executing on a multi/many-core target. In this paper, we focus on the longitudinal flight controller design, his goal is to track accurately altitude, vertical speed and airspeed commands (resp. hc, Vzc and Vac). The controller design is divided into two parts. First, the environment simulation part represents the real system that is to be controlled, that is the aircraft as well as the engines and elevators. Second, the controller part which gathers the control loops (altitude_hold, Vz_control, Va_control) as well as filters. In this paper, we use the ROSACE case study to illustrate how computing WCET for tasks, execution units and also computing the number of data accesses inside each execution unit.

4.2 WCET Task Measurement

As depicted on the controller design in [12] and according to the case study source code, each block of the Simulink controller design represent a task. Therefore, to get the WCET of each block using OTAWA tool, we proceed as follows:

- We have separated the code of each task in a separated file.

Task	WCET (cycle)	First E.U (cycles)	Data access number	Second E.U (cycles)	Data access number
Engine	2185	410	1 read & 1 write	1775	4 read & 2 write
Elevator	7080	75	2 read & 2 write	7005	6 read & 3 write
Aircraft_Dynamics	52160	31280	35 read & 16 write	20880	14 read & 16 write
Vz_Control	3815	2815	9 read & 1 write	1000	2 read & 1 write
Altitude_Hold	3475	515	3 read & 1 write	2960	5 read & 2 write
Va_Control	4430	3125	10 read & 1 write	1305	3 read & 1 write
H_Filtre	5450	4035	8 read & 4 write	1415	3 read & 3 write
Az_Filtre	2665	1250	5 read & 5 write	1415	5 read & 3 write
Vz_Filtre	3875	1720	4 read & 5 write	2155	5 read & 3 write
Q_Filtre	2665	1250	5 read & 5 write	1415	5 read & 3 write
Va_Filtre	5180	5140	9 read & 6 write	40	2 read & 2 write

Table 1: WCET Measurement

- As we have a shared data between tasks, to make computations, it was necessary to set the value of shared parameters for each task.
- We have targeted the ARM architecture and specially the ARM-linux-gnueabi-gcc compiler.
- After the compilation of the task's code, for each task, we have build a flow fact file (*.ff) to identify loop headers using the mkff utility. Called on the executable, it generates a template (*.ff) file where the loop counts are replaced by ? marks.
- Using owcet instruction which allows to use WCET computation scripts dedicated to a specific microprocessor model, we get the WCET of each task of the controller.

4.3 WCET Execution Unit Measurement

Once we have the WCET of tasks, we can go further by calculating the WCET of execution units set within each task of the system. For each we define two execution units, the division is made according to the source code data, the first execution unit processes the input data and the second provides the output data. Then we generate the WCET by following the same OTAWA steps explained above, but for the execution units this time. We also determine the number of data accesses in each execution unit. The Table 1 summarises the results found for each task of the controller system.

5 CONCLUSION

This article presents an approach to extend the classical real-time task model by decomposing task executions into smaller units. We present an approach to extend the current task model in the Cheddar scheduling analyzer with information of internal executions and data accesses. Our model can also be described in AADL. To show that it is possible to acquire data for our proposed model, we experiment with the ROSACE case study and the WCET analysis tool OTAWA.

As a short term perspective, we plan to extend this case study with more relevant measures by investigating different strategies to decompose the tasks into execution units. Future work will also focus on the dynamic control of the operations performed on the data then update the actual simulator in Cheddar to highlight possible deficiencies.

REFERENCES

[1] Ill-ham Atchadam, Laurent Lemarchand, Hai Nam Tran, Frank Singhoff, and Karim Bigou. 2020. When security affects schedulability of TSP systems: trade-offs observed by design space exploration. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 369–376.

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 35–46.

[3] Thomas Carle and Hugues Cassé. 2018. Reducing timing interferences in real-time applications running on multicore architectures. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. 1–11.

[4] Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. 2012. YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms. In *WATERS 2012*. Italy, 21–26. https://hal-upec-upem.archives-ouvertes.fr/hal-00691985

[5] Younès Chandarli, Manar Qamhieh, Frédéric Fauberteau, and Damien Masson. 2014. *Yartiss: A generic, modular and energy-aware scheduling simulator for real-time multiprocessor systems*. Ph.D. Dissertation. UPE LIGM ESIEE.

[6] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. 2014. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 6–p.

[7] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. 2004. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *IFIP World Computer Congress, TC 2*. Springer, 3–15.

[8] Christian Fotsing, Frank Singhoff, Alain Plantec, Vincent Gaudel, Stéphane Rubini, Shuai Li, Hai Nam Tran, Laurent Lemarchand, Pierre Dissaux, and Jérôme Legrand. 2014. Cheddar architecture description language. *Lab-STICC technical report* (2014).

[9] ISO 10303-11 1994. *STEP Part 11: EXPRESS Language Reference Manual*. ISO 10303-11.

[10] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[11] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. 2016. A closer look into the aer model. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 1–8.

[12] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. 2014. The ROSACE case study: From Simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 309–318.

[13] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 269–279.

[14] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. 2004. Cheddar: a flexible real time scheduling framework. In *Proceedings of the 2004 annual ACM SIGAda international conference on Ada*.

[15] Hai Nam Tran, Frank Singhoff, Stéphane Rubini, and Jalil Boukhobza. 2016. Cache-aware real-time scheduling simulator: implementation and return of experience. *ACM SIGBED Review* 13, 1 (2016), 22–28.

[16] Richard Urunuela, Anne-Marie Déplanche, and Yvon Trinquet. 2010. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In *15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. IEEE.

Towards Demystifying Cache Interference on NVIDIA GPUs

Tyler Yandrofski
The University of North Carolina at Chapel Hill
tylerdy@cs.unc.edu

Jingyuan Chen
The University of North Carolina at Chapel Hill
leochanj@cs.unc.edu

ABSTRACT

Obtaining and bounding real-time tasks' worst-case execution times (WCETs) on NVIDIA GPU-based systems is challenging because the details of hardware and drivers are considered proprietary and lack documentation. This makes it difficult to predict the effects on WCETs of interference caused by contention for access to shared caches. This paper describes efforts to demystify, and thus possibly avoid, cache interference effects on NVIDIA GPUs by generating and measuring cache contention.

Keywords

GPUs, CUDA, cache interference

1. INTRODUCTION

Autonomous vehicles have received considerable attention across the real-time system community. Recent advances in computer-vision algorithms and graphics processing units (GPUs) have made fast and intelligent autonomous control systems possible. However, predictability in WCETs remains elusive and makes the certification of real-time safety in autonomous vehicles extremely challenging.

One major requirement for WCET predictability in safety-critical autonomous vehicles is the analysis of latency bounds. Such analysis requires careful consideration of interference from various sources, including cache, DRAM, bus, etc. However, on most NVIDIA embedded systems (currently the dominant platforms for autonomous vehicles) necessary information about the underlying hardware is hidden. Moreover, the memory architecture on GPUs could differ dramatically from that on CPUs, making traditional memory isolation techniques less suitable. Existing WCET estimations for GPU-based real-time systems are hampered by a limited understanding of cache and memory interference, potentially undermining the safety of these systems. The work described here addresses this issue by conducting a thorough investigation of cache interference on NVIDIA GPUs.

Organization This paper is organized as follows. We provide a brief overview of NVIDIA GPU and CUDA programming in Sec. 2 and discuss related work in Sec. 3. We then discuss our methods of efficient generation of extreme cache contention in Sec. 4 and conclude in Sec. 5.

2. BACKGROUND

2.1 GPU and memory architecture

We used the NVIDIA Jetson TX2 in this work. The Jetson line is marketed for "autonomous everything" and is commonly used in embedded applications. The TX2 has an integrated GPU, which shares DRAM with the CPU (as opposed to discrete GPUs with separate DRAMs common in gaming applications). The TX2 also contains a six-core heterogeneous ARMv8 CPU and a shared 8 GB DRAM. The GPU contains two streaming multiprocessors (SMs), each with a private L1 cache and 128 cores. The two SMs share an L2 cache which leads to contention and interference. Only two parameters of the L2 cache are documented – the total size (512 KB) and cache line size (128 bytes). We assume this cache is some variant of a set-associative cache [5].

2.2 CUDA basics

CUDA is a C/C++ extension for parallel programming developed by NVIDIA that allows programmers to execute *kernels*, programs to be executed on the GPU. Executing a kernel involves calling the kernel function with parameters expressing how GPU hardware threads will process the input data in parallel. Threads are organized into blocks, each of which is executed on a single SM. The number of blocks and number of threads per block is specified as parameters by the programmer. The threads in each block are scheduled on the SM in groups of 32 threads, called *warps*. Each thread in a warp executes the same instruction on different data (SIMD). When an instruction stalls on a memory access, the entire warp is context switched for a non-stalled one.

3. RELATED WORK

3.1 Managing cache interference on CPUs

Several works have provided methods to account for CPU cache and memory interference in real-time systems. [3] proposed a framework to profile memory access patterns of safety-critical tasks and a deterministic cache allocation framework to improve the predictability of the system. [6] introduced an analysis method that computes upper bounds of task delays due to memory contention. [7] presented a coordinated cache and memory bank coloring framework that reduces inter-task memory interference on multicore systems. These methods may not apply directly to GPU-based systems, where the computation models are fundamentally different and important architectural details are hidden.

3.2 Dissecting GPU hardware

[2] gave an extensive study on the hardware details for NVIDIA Volta architecture GPUs. [4] introduced a fine-

Table 1: Memory access time baselines (clock cycles)

Scenario	Mean	Minimum	Maximum	S.D.
Cache miss	519	461	1112	72
Cache hit	116	92	128	5

grained micro-benchmarking method to investigate the GPU memory hierarchies on the Fermi, Kepler and Maxwell architectural generations of NVIDIA GPUs. These works mainly measured unknown hardware parameters. In contrast, [1] conducted reverse engineering to unveil the details of the L2 cache and DRAM bank mapping functions on NVIDIA Pascal and Volta architectures.

While these works provide useful information about cache and memory behavior of NVIDIA GPUs, our work aims for a deeper understanding of how cache interference influences WCET predictability. We aim to understand not only how to prevent cache interference, but also how to produce it, so that we may bound WCETs of an arbitrary real-time task.

4. GENERATING CACHE CONTENTION

In this section, we discuss methods for generating extreme cache contention using an application-agnostic kernel that aims to evict all cache lines in the shortest possible time.

Motivation As discussed in the introduction, understanding cache contention effects on WCETs is necessary for the certification of safety-critical real-time tasks in autonomous vehicles. We also aim to inform real-time programmers about application patterns of memory access that lead to cache interference. Our goal is to design a cache-eviction kernel that effectively simulates extreme cache contention with minimal impact on GPU computation resources. We expect it to 1) evict enough cache lines to force most application memory accesses to go to DRAM and 2) run concurrently with any application, evicting efficiently according to the memory accessing behaviors of the system.

Eviction kernel design. The design must consider how kernel memory references result in cache accesses. The GPU L2 cache is used when kernels reference GPU global-memory virtual addresses. The addresses are mapped by hardware to physical addresses in the DRAM shared with the CPU. The physical addresses are then mapped by the cache to *sets* in the L2 cache, each of which can hold multiple cache line *ways*. The details of how physical addresses are mapped to cache sets are not publicly available. We initially assumed a linear mapping where the cache set is indexed by a set of k sequential low-order bits of the physical address [5].

We designed an eviction kernel to access GPU global-memory addresses that map to different L2 cache lines using *pointer chasing*. The kernel iterates through a global-memory array of unsigned integers equal in size to the cache. Each element in the array points to the index of the next element to be visited. Ideally, by visiting elements that map to the first word of each cache line, the kernel can evict every line in the cache.

4.1 Experiment designs

Our first step was to verify that the kernel that is successful in simple, ideal scenarios and simulations. We discuss the settings of these experiments below.

Time measurement. We measured times for memory accesses and kernel execution by reading the per-SM clock64 cycle-counting register. We stored the recorded times in the non-global per-SM memory shared by executing blocks and copied them to global memory later. We first recorded the memory access time for both warm and cold cache reads as baselines. We wrote a kernel that sequentially iterates through a cache-sized array and records the cold access times. Then the kernel iterates again and records the warm access times. Since the second pass could include cache misses, we discarded measurements in the second pass which fell into the cycle range observed to be associated with cache misses for the cold baseline. The observed mean time was 519 cycles for cold accesses and 116 cycles for warm accesses as shown in Table 1. Note that about 5% of cold accesses require more than 593 cycles. The reason for such occasionally large miss times is currently not clear. Potential explanations include DRAM row-buffer conflicts, bus contention, etc.

Run time configuration. To allow efficient eviction, we allocated multiple warps to the eviction kernel so that warp stalls for cache misses could be hidden by the warp scheduler. Also, to explore the effect of memory access patterns, two versions of the eviction kernel were written: one iterates through the arrays sequentially, the other randomly.

Potential problems. We assumed contiguous global memory maps to contiguous physical memory and physical addresses map linearly to cache sets. If these assumptions are violated, an array could contain more addresses that map to certain cache sets than to others. As a result, rather than accessing every cache set once for every cache line in it, the kernel would access some cache sets too few times (and others too many), leaving some cache lines not evicted. To explore this effect, we analyzed the cache coverage of arrays in different parts of memory using multiple pointer-chasing arrays.

4.2 Effective cache eviction

In this section, we explore methods to allow the eviction kernel to effectively evict all cache lines.

4.2.1 Warm cache experiments

Design. We first test our assumptions that contiguous virtual addresses are mapped to contiguous physical addresses and that the function that maps physical addresses to cache lines is a linear index. Under this assumption, each contiguous group of array elements aligned within a cache line should be mapped to a different cache line without any conflicts. Therefore, if a kernel iterates this array twice, all reads in the second pass should experience cache hits - full cache coverage.

We designed an eviction kernel that performs random pointer chasing over 8 arrays of 512 KB located at disjoint memory locations. For efficient execution, the kernel runs on 8 warps. The kernel iterates an array twice in a row, recording the time for each memory access in the second pass, before repeating for the next array. If our assumption holds, all access times should be consistent with the baseline for warm accesses. Figure 1 summarizes the results of multiple runs of the kernel for a sequential access pattern. A random access pattern yielded similar results.

Observations. The typical elapsed time for cache hits is in the range of 104 to 128 cycles[1], while the typical time for

[1]25% of accesses took 88 or 89 cycles because the nvcc compiler unrolls the loop into four PTX assembly sequences, the last of which is structured in a way that delays the memory access stall until after the access time is recorded

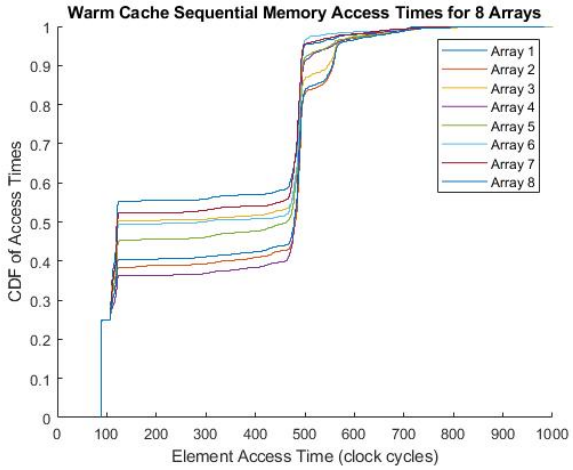


Figure 1: Cumulative distribution function plot of memory access times over 8 different sequential pointer chasing arrays.

cache misses is at least 450 cycles. These ranges are consistent with our baseline ranges. However, as shown in the figure, only 36% to 55% of the accesses are within the hit latency range, depending on the array. Thus, far from 100% of the cache was covered by accessing the cache-sized array - 36% in the worst case. The low coverage and variation in hit rates indicate our assumption does not hold. Non-contiguous physical memory or a non-linear hash function is used, which causes many physical addresses in the pointer chasing array to share cache lines.

4.2.2 *Future work - Reverse engineering*

Without low-level knowledge of the physical memory and cache mapping functions, it's almost impossible to achieve ideal cache coverage. Therefore, we plan to reverse engineer the mapping function of TX2's L2 cache. Obtaining the exact hash function would allow us to evict arbitrary cache lines with simple bit operations. Besides reverse engineering, we also aim to achieve optimal allocation of physical addresses for eviction, so that the eviction kernel may cover all cache lines without allocation of unrealistically large arrays or fragmented chunks of physical memory.

4.3 Concurrent eviction

In addition to evicting the entire cache when executed alone on the GPU, the eviction kernel must be able to stress a real victim kernel's memory accesses. We tested two methods of executing both kernels concurrently on the same GPU: 1) running in multiple processes and 2) running in different threads of the same process.

4.3.1 *Multi-process experiment*

Design. Our first experiment evaluated an application kernel's performance when executed concurrently with an eviction kernel in a second CUDA context (process). Note that in CUDA if multiple contexts share the GPU, only one context can access it exclusively at a given time. The victim application is a memory-intensive kernel that reads every integer in a cache-sized array (131,072 integers) sequentially and sums them. It runs with one block and one warp per block. The eviction kernel is the one from 4.2.1 but with a random access pattern. Two different thread layouts are used: 1 block-1 warp per block and 4 blocks-32 warps per block (the max threads available to the TX2). We ran the application in three scenarios: alone as a baseline, concurrently with a "compute-only" kernel in a second process, and concurrently with a true eviction kernel in a second process. The "compute-only" kernel simply executes an infinite loop without accessing memory. In this case, we can measure the effect of multi-process concurrent execution separately from the effect of cache interference.

Table 2: Application execution times - multi-process

Opposing kernel	Mean (ms)	Max (ms)	S.D. (ms)
None	21.08	23.71	4.59
Compute 1x1	44.67	50.19	6.71
Compute 4x32	46.22	51.25	6.80
Eviction 1x1	44.69	50.18	6.71
Eviction 4x32	46.24	52.49	6.80

Observations. Table 2 summarizes 4096 samples of the application kernel's execution times per scenario. The "None" scenarios mean the application runs alone as a baseline. While the execution times were much higher with the compute-only kernel than with the baseline, the compute-only and eviction kernel results are similar. This indicates that the application process experiences increased execution times when executing concurrently with another process, but that the eviction kernel's memory accesses had little effect.

Discussion. Both results can be explained by considering that multi-process execution in CUDA is achieved by time-sliced execution, rather than real concurrent execution. Due to time-slicing and context switches, the execution times will approximately double when the application kernel shares the GPU with other contexts, as it did with the compute-only kernels (2.12x and 2.19x increase compared to application-only). Only memory accesses that re-establish the application's working set at the start of its time slice were susceptible to eviction-triggered cache misses. This working set interference was overwhelmed by the 2x time-slicing effects.

4.3.2 *Single - process experiment*

Design. Since time-slicing effects limit interference between processes, we next explored cache interference between kernels in a single context (process). We executed a victim application that randomly accesses elements in an array of cache size (131,072 integers) as a thread (*pthread*) in the process. Another thread in the same process ran the eviction or compute-only kernels from 4.3.1, but with a 2 block-32 warps per block thread layout. By allocating 64 warps to the eviction kernel (as opposed to 1 application warp), we minimize the effect of stalls due to cache misses since many other warps are available to schedule in place of one that stalls. Additionally, each kernel ran on a dedicated SM, avoiding interference effects caused by sharing cores with the victim. As before, we ran the application alone, with a compute-only kernel, and with a true eviction kernel.

Observations. Measurements of 4096 samples for each setup are summarized in Table 3 in the entries labeled "R" for random application kernel (the sequential "S" entries are described in 4.4.1). "None" scenarios correspond to the application kernel running alone. This time, the application-only measurements are similar to those for the compute-only se-

Table 3: Application execution times - single-process

Scenario	Mean (ms)	Max (ms)	S.D. (ms)
None-R	30.72	33.27	0.04
None-S	17.15	19.49	0.04
Compute-R	30.97	33.15	0.04
Compute-S	17.36	19.43	0.03
Eviction-R	64.59	64.81	0.06
Eviction-S	23.25	23.35	0.03

tups, a sensible result since there were no competing warps on the application's SM. This indicates that the execution times were not affected by GPU scheduling policies so any increase in application execution times can be attributed solely to the memory accesses of the eviction kernel.

Discussion. The results show that significant cache interference between kernels in the same CUDA context is possible, although our methods in this experiment are impractical for two reasons. First, allocating many warps to the eviction kernel and few to the application is unrealistic, especially since an entire SM was dedicated to the eviction kernel. More work is needed to determine a balance of execution resources that creates enough cache interference for accurate WCET measurement without unduly restricting application performance. Second, we executed the kernels in a single process by running both programs as threads, which is not viable for every potential victim application since it requires significant modification to process-based applications.

4.3.3 Future work - MPS

We plan to use CUDA Multi-Process Service (MPS) to solve the second problem without introducing the time-sharing effects observed in 4.3.1. MPS is an implementation of CUDA for truly concurrent execution of multiple processes' GPU kernels within a single context managed by the MPS server. Future work is required to implement the experiments from this section using MPS and confirm that sufficient cache interference is possible using this approach.

4.4 Eviction-application interaction

In addition to ensuring full cache coverage and efficient concurrent execution, it's important to consider how characteristics of application memory access patterns might impact cache interference. Our next experiment considers sequential application data accesses.

4.4.1 Sequential vs. random application accesses

Design. Our scenario was the same as in the previous experiment (1 warp application kernel and 64 warp eviction kernel), but with the application making sequential accesses rather than random ones.

Observations. Table 3 compares the results for this experiment (sequential application labeled with "S") and the previous one (random application labeled with "R"). Note that the execution times for sequential access scenarios are much smaller than those for random access, both with and without the eviction kernel executing.

Discussion. The observed results can be explained by the presence of a read-ahead effect in sequential memory accesses. A cache line that has already been buffered in the SM due to a previous read will have a lower access time than if it had required a cache or DRAM access. This effect would reduce cache interference for any sequential memory access pattern with read-ahead.

4.4.2 Future work

In future work, we will evaluate the effects of other memory access patterns on the interaction between eviction and victim kernels. For example, applications with smaller working sets may prove to be difficult to evict quickly enough to create interference. Certain distributions of application data in physical memory may make interference easier by concentrating data in locations that map to certain cache sets. The eviction kernel pattern may matter as well, for instance evicting an entire cache set at a time versus rotating between lines from different cache sets. It may also be possible to focus eviction efforts on cache sets that are high-frequency for a specific application.

5. CONCLUSIONS

In this paper, we discussed challenges in generating extreme GPU cache interference and presented our current experiments and potential future work to achieve powerful cache eviction kernels. By gaining a more detailed low-level understanding of the GPU cache architecture and available concurrency options, a more profound analysis of the impacts of cache interference on GPU-based systems, including case studies on real workloads, will be possible in the future.

6. REFERENCES

[1] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019.

[2] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[3] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013.

[4] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[5] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[6] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[7] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692. IEEE, 2013.

A Traffic Infrastructure-Enabled Task Scheduling in Vehicular Edge Computing Networks

Pratham Oza
prathamo@vt.edu
Virginia Tech

Thidapat Chantem
tchantem@vt.edu
Virginia Tech

ABSTRACT

Large-scale deployment of connected vehicles and traffic infrastructure along with the advances in vehicle-to-everything (V2X) technology has paved the way for novel automotive applications to enhance safety and driving experience. Though, such applications require increased computational capabilities with timeliness guarantees. To satisfy the resource demands, the vehicular edge computing (VEC) paradigm extends the computational abilities of the vehicles by enabling them to offload certain tasks onto the edge servers deployed at the road-side units (RSUs). However, the real-time performance of the VEC tasks heavily depends on the network coverage and vehicle mobility, especially in urban traffic scenarios. In this work, we motivate the requirement of a real-time task model and task offloading schemes that incorporate the readily available traffic flow and signal timing data to guarantee real-time performance of tasks offloaded on the edge.

1 INTRODUCTION

Advancements in wireless technology, vehicle-to-everything (V2X) connectivity, artificial intelligence and sensors have led to the proliferation of smart vehicles and intelligent traffic infrastructure. By leveraging on-board sensors such as cameras, LiDARs and RADARs, as well as the data aggregated from surrounding connected vehicles and traffic infrastructure, vehicles can now provide enhanced safety and efficiency. Similarly, by utilizing smart road-side units (RSUs) and relevant information from vehicles, the traffic infrastructure can enable efficient traffic control with minimal delays, prioritized emergency vehicle movements, dissemination of safety-critical messages, etc. [8]. V2X connectivity is also being used to provide media-rich infotainment solutions such as augmented reality (AR) and video streaming, to the vehicle users to improve the overall driving experience [3]. However, such data-intensive applications are accompanied with increased computational requirements where on-board processing units located in the vehicles are not sufficient to satisfy such increased demands.

Cloud computing offers centralized remote servers that the vehicles can utilize, instead of the on-board processors, to perform certain computationally intensive tasks such as, AR-based heads up displays (HUDs) for real-time safety warnings or natural language processing (NLP) based driver command cognition systems. Along with real-time processing, such media-rich applications also call for low-latency real-time communication, which the centralized cloud architecture fails to provide, leading to poor quality of service (QoS) [12].

As the number of RSUs are continuously increasing to provide seamless V2X connectivity, enhanced traffic detection, and traffic measurements, the vehicular edge computing (VEC) paradigm proposes deployment of cloud services on these RSU to bring computation closer to the vehicles [4]. VEC offers reduced communication latency along with increased computational capabilities and thereby satisfying both real-time processing as well as communication constraints. Through V2X technologies such as dedicated short range communication (DSRC) or C-V2X, vehicles can now *offload tasks* that are computationally-intensive and/or delay-sensitive, onto these RSUs to utilize the additional processing resources at low communication latency.

While RSUs offer additional computational resources, with increasing number of vehicles especially in urban areas, task offloading policies are necessary to manage the resources. Recent works have proposed various schemes of distributing the tasks on the RSUs within range, by maximizing the resource utilization of the system [3], minimizing the energy expenditure of the vehicle [12], or by maximizing the QoS for all road users [13]. However, these schemes lack the consideration of criticality and timeliness requirements of the tasks that are being offloaded on the edge.

Further, due to limited coverage area of the RSUs, the communication link between a vehicle and an RSU is interrupted as a vehicle moves out of coverage [11]. This indicates that the task offloading schemes must also account for vehicle mobility as well as the transmission and handover/ takeover time between the vehicles and the RSUs while allocating resources, to ensure that the tasks are offloaded and completed in time [7]. Most VEC task offloading schemes that incorporate vehicle mobility consider an idealised traffic pattern where vehicles move along a constant stream of straight moving traffic, as seen in highway-like driving [5]. But, in urban traffic scenarios, vehicle mobility is heavily impacted by intersections and dynamic traffic density, which in turn affects VEC resource management [7]. Urban driving also offers increased RSUs with multiple sensors, cameras, traffic lights and traffic controllers deployed across the road network which can be utilized to provide seamless connectivity and meet the resource requirements for the increased traffic demands.

Most state-of-the-art applications in traffic infrastructure are utilizing edge-based resources and data aggregated through RSUs to provide *temporal predictability* in traffic movements and traffic signal control in an urban road network [8, 10]. A closely related work [7] models task offloading for vehicles in an urban VEC environment with signalized intersections as an RSU-vehicle matching problem using car-following models to estimate mobility. However, they do not utilize the traffic flow data to provide a predictable offloading schedule. In this work, we propose using the predictive traffic behavior from the traffic flow data to influence the scheduling of tasks and allocation of resources in a VEC framework. Specifically, we present a task offloading scheme for opportunistically scheduling tasks on the available RSUs, by considering the

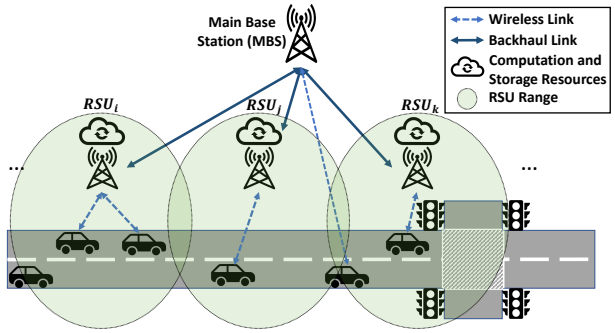


Figure 1: A typical vehicular edge computing framework

traffic signal timings and the wait times of the vehicles at multiple intersections.

Along with vehicle mobility and traffic flow parameters, the VEC task scheduling framework must also be aware of the properties of the tasks such as temporal requirements and deadlines, task criticality, effect on quality-of-service, etc. However, the recent work [5, 7, 9] lack a generic task model that can represent realistic workloads in the vehicular edge environment. Through this work, we also emphasise on the requirement of a real-time task model that characterizes a practical workload containing time-sensitive, critical tasks as well as delay-tolerant tasks with some effect on the quality of service of the system.

In all, this work highlights the need for a real-time model to represent VEC networks along with a task offloading scheme that:

- is deadline-aware and can cater to time-sensitive safety-critical tasks while providing timeliness guarantees for computationally-intensive data-rich tasks,
- incorporates readily available traffic data at intersections in modelling vehicle mobility through urban road networks to accuractely depict the delay overheads to VEC tasks, and,
- relies on a VEC task model that replicates realistic workloads in an urban environment.

Next, we will discuss the system model for the VEC framework and a motivating example of the advantages of incorporating the knowledge of traffic signal timings in task offloading process.

2 SYSTEM MODEL

2.1 VEC architecture and its components

Consider an urban traffic environment enabled with vehicular edge computing components distributed along the RSUs located by the road-side (Figure 1). The vehicles traveling along the road network utilize the processing resources at the RSUs equipped with VEC servers to meet their computation demands. The RSUs tend to have a smaller coverage area to enhance data transmission speeds. Coverage areas of consecutive RSUs have minimal overlap to expand the range of connectivity and reduce duplication of resources and communication. The VEC servers located with the RSUs provide increased computational capabilities with faster processing speeds as compared to the vehicles' on-board processors. RSUs are often

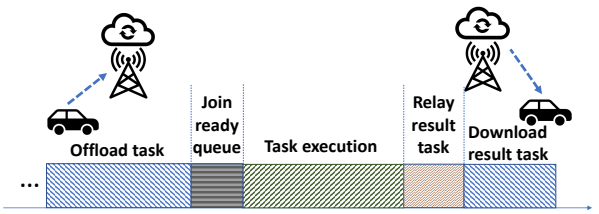


Figure 2: A task execution cycle

equipped with multiple-input and multiple-output (MIMO) technology to communicate with multiple vehicles within its coverage area, at once.

A main base station (MBS) located along the urban network provides connectivity to the mobile devices, RSUs as well as the vehicles, and have a much larger coverage area encompassing multiple traffic intersections. Within the VEC paradigm, the MBS acts as a central coordinator that accepts offloading requests from the vehicles and schedules them on the RSUs for processing as per the underlying offloading policy. The MBS is connected with the RSUs through a wired (optical fiber) link that ensures stable network with reduced data transmission latency [4].

As the vehicles drive through the urban road network, they require the edge resources to compute multiple tasks originating from data-intensive applications or time-sensitive safety-critical applications. The vehicles require the edge resources to execute the tasks when either it lacks the on-board computational capability to complete the task execution or the tasks require data from other connected entities (vehicles and/or infrastructure) that is only available at the edge. These tasks have varying execution times and criticality, depending on the application. The vehicle sends the task information to the MBS after which the MBS assigns an appropriate RSU to execute the task as per the offloading policy. Further, the vehicles abide by the standardozed traffic rules and have ADAS features on-board to follow driving models such as Intelligent Driver Model (IDM) [6] for safe urban driving. The urban environment also consists of multiple signalized intersections through which the vehicles traverse, and are controlled using traffic lights that follow a green-yellow-red pattern. Novel traffic control techniques employ a centralized traffic controller deployed at the edge server to estimate the traffic flow and calculate the signal timings [8, 10]. With the VEC architecture and its individual components established, as shown in Figure 1, we will now discuss the need for accurately estimating vehicle mobility to guarantee real-time performance in the VEC environment.

2.2 Traffic mobility estimation for performance guarantees

When a vehicle requests the MBS for an edge resource to perform certain tasks, there are multiple network-related constraints that the MBS must consider before assigning an RSU resource to a vehicle. Note that the RSUs and the MBS are connected through a wired network and the MBS has the knowledge of the resource utilization of all RSUs within its coverage area.

- **Offloading task from a vehicle to an RSU:** A vehicle needs to be within the coverage area of the RSU before it can begin

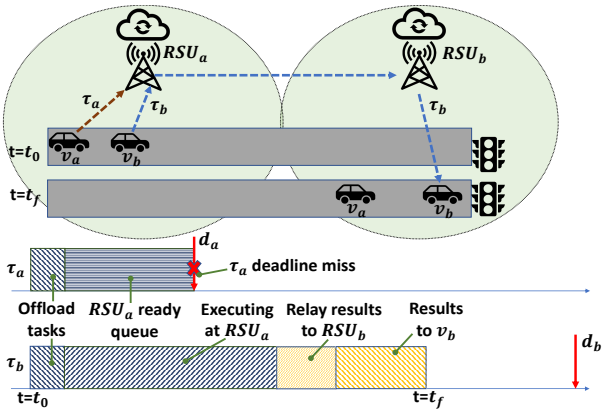


Figure 3: Traffic data unaware task offloading schemes

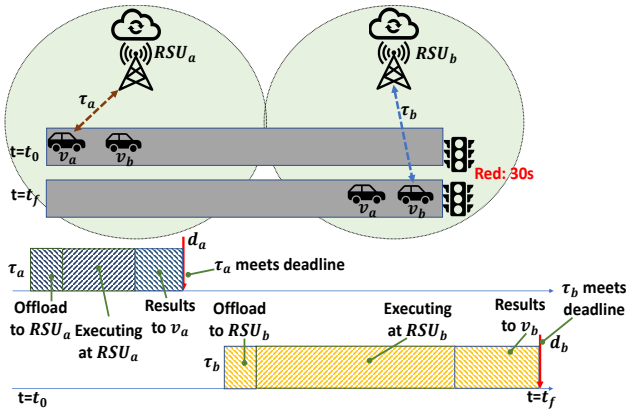


Figure 4: Proposed traffic data and deadline-aware task offloading scheme

offloading the task. The MBS must account for the time required for the vehicle to travel to the coverage area of an RSU before it can begin the offloading process.

- **Continuity of task offloading:** Once the vehicle is in the coverage area and starts offloading the task onto the RSU, the time it takes to offload a task depends on the size of the task. The larger the size, the longer it will take to offload the task. However, the vehicle is still in motion and may exit the coverage area of the RSU before the offloading completes. The RSUs then need to relay the already offloaded task information to subsequent RSUs while the vehicles continue offloading the remaining data to the next RSU. This incurs additional delays in re-transmission of data between the RSUs.
- **Wait time before execution:** Once the task is successfully offloaded onto the RSU, it joins the ready queue and begins execution only when the computation resource is available.
- **Offloading task from an RSU to the vehicle:** Once the task completes execution the resulting data must be offloaded back to the vehicle. Again, since the vehicle is already in motion, it may be the case that the task execution took place at an RSU which is away from the vehicle. The results must be transmitted to an appropriate RSU whose coverage area contains the vehicle currently, before the results are actually offloaded to the vehicle. The time delay associated with this transmission too, requires precise knowledge of the vehicles location and mobility.

Note that the transmission times and the delays mentioned here are over and above the actual task execution time at the edge resource (Figure 2). Clearly, all network-related delays require accurate estimation of vehicle mobility. In a highway setting, estimation of this time is trivial since the traffic flow is mostly constant. However, in an urban network, the travel time is not only influenced by the vehicle's own motion but is also impacted by the traffic lights and the traffic density through the road network. We therefore propose integrating the available traffic data and traffic signal timings for modeling the vehicle motion to more precisely estimate this travel time delay. We show in Section 3, that incorporating traffic data and signal timings to estimate vehicle's mobility and location helps in maximizing the resource utilization of the entire VEC system with more vehicles able to offload their desired tasks on the VEC network.

2.3 A task model for VEC workloads

Most existing task offloading policies [7, 12] are best-effort and do not consider task deadlines while scheduling and offloading tasks over the RSUs and hence provide no real-time performance guarantees. However, multiple standardization groups (European Telecommunications Standards Institute (ETSI) and the 3GPP), highlight the necessity for performance guarantees in VEC especially for safety-critical applications [2]. To ensure timeliness guarantees within the VEC paradigm, it is not only important to accurately estimate the communication overheads which rely heavily on vehicle mobility, but it is also necessary to consider task deadlines, especially for safety-critical tasks.

We denote the j^{th} task to be offloaded from i^{th} vehicle as $\tau_{i,j}$ which is represented by the tuple $\{b^o_{ij}, C_{ij}, d_{ij}, b^d_{ij}\}$ where,

- b^o_{ij}: size of the task (in Mb) to be offloaded from the vehicle to the RSU,
- C_{ij}: execution time of the task on the RSU,
- d_{ij}: absolute deadline of the task, and
- b^d_{ij}: size of the result (in Mb) to be offloaded from the RSU to the vehicle.

The existing works do not have a generic task model to represent workloads in VEC [7, 9, 12]. As explained in Section 2.2, the transmission times and communication delays depend on the size of the task and the computed results, along with the vehicle mobility. Hence, we want to emphasize the necessity of a real-time model to represent realistic workloads and provide novel scheduling schemes that enhance the predictability of the VEC environment and adhere to the performance requirements and timeliness guarantees standardized by the industry [2].

3 MOTIVATION AND CHALLENGES

As discussed in the previous section, the main base station (MBS) is a central entity to coordinate and schedule tasks on to the RSUs such that the delays involving data transmission to/from the RSUs as well as the execution time are accounted for while ensuring that the tasks meet their deadlines. The delays rely heavily on the MBS' estimate of the vehicles' location and its mobility. Previous works such as [3, 11, 13] do not consider dynamic traffic mobility especially in

an urban environment where the demand for edge-based resources is high. [7] considers urban mobility through road networks with multiple intersections but does not incorporate traffic signal timing data in providing predictable offloading strategy. In all, none of these approaches guarantee real-time performance and are best-effort in either minimizing energy utilization, transmission costs, or transmission delays.

Clearly, the total execution times for the tasks to be offloaded depend heavily on the vehicle's mobility. Further, multiple safety-critical applications with strong reliance on cooperative edge-based processing, such as merging at blind turns, collision-free intersection management, etc. require hard deadline guarantees [1].

To motivate our work, let us consider a simple vehicular edge environment as shown in Figures 3 and 4 in which two vehicles, v_a and v_b require the RSUs to execute certain tasks. Here, v_a needs to offload a time-critical task, τ_a with a deadline d_a, and v_b needs to offload a data-intensive task, τ_b with a deadline d_b. The execution time of τ_a is shorter than that of τ_b. Additionally, there are two RSUs in the vicinity namely, RSU_a and RSU_b, which have the resource bandwidth of only performing one task at a time. Both v_a and v_b are approaching a signalized intersection. As mentioned in Section 2, both tasks can be uploaded to an RSU at once due to the MIMO functionality.

Task execution with state-of-the-art offloading scheme: Now, as shown in Figure 3, a task offloading scheme such as [7], which only considers a simplistic car-following model for vehicle mobility and neither utilizes exact traffic information, nor is it deadline-aware and hence offloads tasks τ_a and τ_b on RSU_a. Task τ_b starts execution while τ_a joins the ready queue. The state-of-the-art approaches assume a FIFO scheduling of the tasks. Since τ_b is a data-intensive task, it takes longer time to execute, leading to a missed deadline for a safety-critical task, τ_a. Such missed deadlines could hamper the driving and safety performance of the vehicles. Further, v_b moves out of the coverage area of RSU_a and stops at the red light within the coverage area of RSU_b leading to a longer total execution time of τ_b since the task results need to be transmitted to RSU_b from RSU_a and then to v_b.

Task execution with proposed offloading scheme: Now, consider a deadline-aware, traffic infrastructure-aware task scheduling scheme as in Figure 4. The task offloading policy knows that the upcoming traffic light is expected to stay red for 30 seconds, and vehicle v_b is going to stop for a prolonged period due to the red light. The policy therefore offloads only task τ_a onto RSU_a and due to its shorter execution time, it finishes execution and results transmission back to v_a while it is in the coverage area of RSU_a and thereby meeting its deadline. Additionally, even though τ_b is a computationally-intensive task, it successfully completes offloading to RSU_b, execution at RSU_b and transmission of results from RSU_b, all while it is waiting at the traffic light.

Thus, a criticality-aware VEC task model with a traffic infrastructure and deadline-aware task offloading policy does not only maximizes the resource utilization of the RSUs but also reduces the transmission delays between the RSUs, while ensuring that all tasks meet their deadlines.

4 CONCLUDING REMARKS

Through this work, we motivate the need for a real-time model for urban VEC environment with task scheduling policy that,

- exploits the readily available traffic data through RSUs and sensors, as well as the traffic signal timings information through connected infrastructure to predictably offload and schedule tasks on the edge servers,
- provides timeliness and performance guarantees by limiting the deadline misses and maximizing resource utilization, and
- relies on a deadline-aware task model specifically to replicate realistic workloads.

5 FUTURE WORK

We will formulate the proposed task model for realistic workloads that resemble deep-learning based tasks with timeliness and QoS requirements. Using a traffic simulator such as VISSIM, we will emulate large-scale traffic with connected vehicles on an urban roadmap with multiple intersections and traffic lights. This experimental setup will be used to deploy and evaluate our proposed task model and VEC offloading scheme.

REFERENCES

[1] S. Aoki and R. Rajkumar. 2019. V2V-based Synchronous Intersection Protocols for Mixed Traffic of Human-Driven and Self-Driving Vehicles. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1–11. https://doi.org/10.1109/RTCSA.2019.8864572

[2] Multi-Access Edge Computing. 2018. Study on MEC Support for V2X Use Cases. *Standard ETSI GR MEC* 22 (2018), V2.

[3] Yueyue Dai, Du Xu, Sabita Maharjan, and Yan Zhang. 2018. Joint load balancing and offloading in vehicular edge computing and networks. *IEEE Internet of Things Journal* (2018).

[4] Hisashi Futaki. 2020. RSU apparatus, base station apparatus, control node, and methods therein. US Patent 10,595,157.

[5] X. Huang, L. He, and W. Zhang. 2020. Vehicle Speed Aware Computing Task Offloading and Resource Allocation Based on Multi-Agent Reinforcement Learning in a Vehicular Edge Computing Network. In *2020 IEEE International Conference on Edge Computing (EDGE)*.

[6] Martin Liebner, Michael Baumann, Felix Klanner, and Christoph Stiller. 2012. Driver intent inference at urban intersections using the intelligent driver model. In *2012 IEEE Intelligent Vehicles Symposium*. IEEE, 1162–1167.

[7] Pengju Liu, Junluo Li, and Zhongwei Sun. 2019. Matching-based task offloading for vehicular edge computing. *IEEE Access* (2019).

[8] P. Oza, T. Chantem, and P. Murray-Tuite. 2020. A Coordinated Spillback-Aware Traffic Optimization and Recovery at Multiple Intersections. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.

[9] J. Sun, Q. Gu, T. Zheng, P. Dong, A. Valera, and Y. Qin. 2020. Joint Optimization of Computation Offloading and Task Scheduling in Vehicular Edge Computing Networks. *IEEE Access* 8 (2020), 10466–10477. https://doi.org/10.1109/ACCESS.2020.2965620

[10] S. Ucar, T. Higuchi, and O. Altintas. 2020. Signal Phase and Timing by a Vehicular Cloud. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*.

[11] Chao Yang, Yi Liu, Xin Chen, Weifeng Zhong, and Shengli Xie. 2019. Efficient mobility-aware task offloading for vehicular edge computing networks. *IEEE Access* (2019).

[12] Zhenyu Zhou, Pengju Liu, Zheng Chang, Chen Xu, and Yan Zhang. 2018. Energy-efficient workload offloading and power control in vehicular edge computing. In *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*.

[13] Chao Zhu, Giancarlo Pastor, Yu Xiao, Yong Li, and Antti Ylae-Jaeaeski. 2018. Fog following me: Latency and quality balanced task allocation in vehicular fog computing. In *2018 15th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*.