Furió Novejarque, C.; Feliu-Pérez, J.; Petit Martí, SV.; Duro-Gómez, J.; Sahuquillo Borrás, J. (2018). A Workload Generator for Evaluating SMT Real-Time Systems. IEEE Computer Society. 367-374. doi:10.1109/HPCS.2018.00067

A Workload Generator for Evaluating SMT Real-Time Systems

Clara Furió, Josué Feliu, Salvador Petit, José Duro, and Julio Sahuquillo
Departamento de Informática de Sistemas y Computadores (DISCA)
Universitat Politècnica de València, Spain
Email: clafuno@upv.es

***Abstract*—Real-time tasks have experience a significant complexity increase in the last years. We can find examples of real-time tasks in nowadays systems that control self-driving cars or multimedia systems, among others. To cope with the high performance requirements of such systems, real-time systems are moving from simple in-order processor to complex out-of-order multicore processors. Furthermore, we expect real-time systems to use simultaneous multithreading (SMT) processors in a near future since these architectures address two key design concerns of embedded systems, that is, they provide higher performance and power efficiency than single-threaded multicores.**

The main drawback that multicores and SMT architectures present from a real-time perspective is that they implement shared resources. Single-threaded multicores usually share the main memory and the LLC, and SMT processor share additionally most of the microarchitectural core resources. Processes running concurrently can interfere in the shared resources, which increases the performance variability and predictability of these systems. We expect an increasing effort in the next years to mitigate these drawbacks and implement real-time systems with multicore SMT processors.

Workload generation is a tedious and time-consuming task in the real-time research field because the workloads dispose of many parameters that should be correctly adjusted to provide flexible and representative workloads. Typically used workload generators, however, fail when designing workloads for theses architectures because they are not aware of the architectural characteristics of SMT systems. In this paper we present the task class-based (TCB) workload generator aimed at providing workloads to evaluate real-time systems with SMT multicore processors in an ease and automatized way.

***Keywords*—Real-time systems; Simultaneous multithreading (SMT); Workload generator.**

I. Introduction

Real-time systems are systems that should produce their outputs within certain time constraints. Unlike other systems such as high-performance computing (HPC) systems, the correctness, or at least the quality of service, of real-time systems does not only depend on the computation of the correct outputs, but also on obtaining these outputs within the defined time constraints.

The computational requirements of real-time systems have been growing during the last decade. Former real-time systems used to perform relatively simple control tasks with tight time constraints. Nowadays, real-time systems carry out multiple tasks that require higher performance such as self-driving cars, multimedia systems, pattern recognition, etc. [1] [2] [3] In some systems, the real-time constraints are strict or *hard*, but other systems such as the multimedia ones can tolerate small percentages of deadline misses, which translates in a small but acceptable reduction in the quality of service.

To reach the required performance, many of these real-time systems are moving from simple processors to current multicore architectures [4] because of their higher performance. Following this trend, we expect them to move to multithreaded cores in a near future since they improve both performance and power efficiency, two main design concerns in embedded systems. The key challenge that multicore and multithreaded processors must address to be used in real-time systems is that they implement resources that are shared among the processes running concurrently. Resource sharing increases execution time variability due to the inter-application interference, which can be partially mitigated with specific hardware and software mechanisms [5]. Therefore, it is expected that these research topics will pay a lot of attention both from the academia and the industry in the near future.

To evaluate a real-time system during the design phase, a wide and representative set of real-time workloads needs to be used. However, the design of real-time workloads is much complex than the design of conventional benchmarks, such as high-performance benchmark suites. These workloads must consider not only the computational component, but also the deadline and period for each task. A workload generator should be able to adjust them in order to facilitate or difficult the fulfilling of certain deadlines. Either tasks whose deadlines can be easily meet or are impossible to meet will not have much interest and will be hardly representative of an actual real-time system. In addition, workload generators such as the UUniFast algorithm that have been typically used to generate workloads for real time systems fail to meet their goal when requiring workloads with an utilization greater than one, which is the case of multicores with SMT cores. To overcome this issue, in this paper we propose the task class-based (TCB) workload generator designed to facilitate and automatize the process of generating workloads to evaluate real-time systems with these architectures.

The rest of the paper is organized as follows. Section II reviews the key concepts of real-time systems and SMT processors. Section III presents the TCB workload generator devised in this work. Section IV describes the experimental framework and Section V discusses the validation of the work-

load obtained with the proposed generator. Finally, Section VI presents some concluding remarks.

II. Background

This section presents key concepts about real-time systems and SMT processors to help understand this work as well as the used terminology.

A. Real-Time Systems

Nowadays, real-time computing plays a crucial role since more and more complex systems rely, partially or completely, on computer-based control. Although advances in computer hardware technology help improve system throughput and increases the computational power, this does not guarantee that the timing constraints of a given application will be met. In fact, whereas the main goal of high performance computing is, in general, to minimize the execution time of the running workload (i.e. the set of tasks) the objective of real-time computing is to meet the individual timing requirements for each task of the workload. In this context, a real-time system is a computer-based system which has to execute their applications within a specified time. In other words, it is not only important that the system provides the correct output, but also the point of time when it does.

A real-time *task* is typically a sequential *process* that is executed in a conventional single-threaded processor or in a hardware context of a multi-threaded processor. From now on, task and process will be used as synonyms. A real-time system must execute a set of recurrent tasks. The set of rules that determines the order in which tasks are executed is coded in the so called *scheduling algorithm*. A task that can be executed on the processor is called an *active task*. A task waiting for the processor is called a *ready task*, whereas the task in execution is the *running task*. All ready tasks waiting for the processor are kept in the so called *ready queue*.

Real-time tasks are characterized by a series of timing constraints. One of the most representative constraint is the task *deadline*, which represents the maximum time before which the task has to complete its execution. Depending on the consequences on the system of a missed deadline, real-time tasks can be classified into hard real-time (HRT) tasks or soft real-time (SRT) tasks. HRT tasks *must* fulfill their deadline. Otherwise, the system integrity could be compromised and the consequences could lead to important damages; for instance, the altitude control in an aircraft. On the other hand, SRT tasks are required to achieve a minimum number of deadlines to guarantee a certain QoS; for example, a minimum number of frames per second in a video streaming. In this case, a late response does not derive in harm but in a deterioration of the user satisfaction.

Each task T_i of a group of tasks or workload $T = \{T_1, \ldots, T_n\}$ is modeled with a series of parameters. In this work, we focus on three main parameters: worst case execution time (WCET), period, and deadline; that is, $T_i = \{WCET_i, P_i, D_i\}$ (see Figure 1):

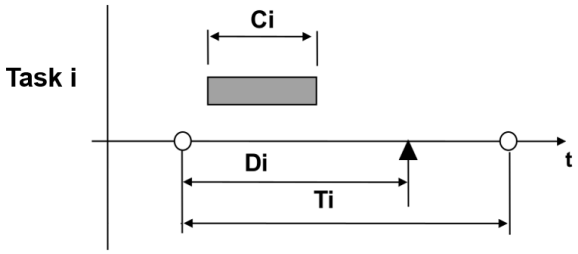


Fig. 1. Real-time task model.

- $WCET$ refers to the execution time in the worst possible scenario and is a critical feature of real-time applications. This time can be approached either by measurement techniques or delimited using analytical techniques. For instance, in case of single-threaded multicore systems where cores only share the last level cache (LLC) and main memory, WCETs can be estimated by running applications concurrently with microbenchmarks that stress these resources.
- P_i is the time interval between two successive activations of the task.
- D_i represents the maximum allowed time for the task to execute. That is, it determines the time at which a task run must be completed with respect to the beginning of each active period. As discussed previously, missing deadlines compromises the QoS on SRT systems and can be equivalent to a complete system failure in HRT systems. Note that, in order to simplify the analysis of real-time systems, it is usually assumed that $D_i = P_i$ [6].

Another important factor in the analysis of real-time systems is the processor utilization factor, which represents the fraction of processor time used by the workload. It is typically obtained using Equation 1, where C_i is the execution time of task i (usually considered equal to the task WCET), and P_i is the period of the task.

$$U = \sum_{i=1}^{N} \frac{C_i}{P_i} \approx \sum_{i=1}^{N} \frac{WCET_i}{P_i}, \tag{1}$$

The utilization determines whether a system is schedulable or not according to the scheduler policy. In other words, it helps analyze if a system would correctly work (i.e. no deadlines are missed) with a specific workload and scheduling algorithm. Moreover, the number of missed deadlines in a given execution is related to the utilization so higher processor utilizations imply more deadline misses and vice-versa.

B. Simultaneous Multithreading

Multicore processors with simultaneous multithreading (SMT) [7] cores are able to execute multiple threads simultaneously in each core. To this end, they implement multiple hardware contexts that maintain the state of each individual thread independently. From the operating system point of view,

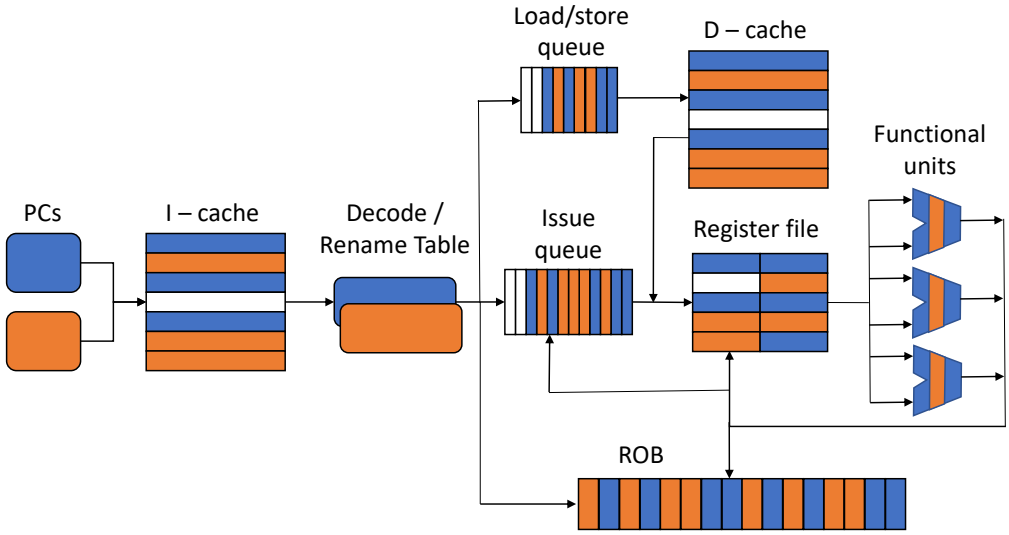


Fig. 2. Pipeline of a generic SMT processor.

one physical SMT core is seen as multiple logical cores, as many as hardware contexts supported by the SMT core.

SMT processors improve throughput with minimal hardware increase over single-threaded processors, enhancing the performance for a given area and power budget [8] [9]. Thus, they have become the *de facto* design choice in current high-performance commercial processors. In this context, due to the convergence among different segments of the processor market, it is expected that these processors proliferate across embedded devices in the near future.

SMT processors replicate the hardware that supports the state of processes for each hardware context. The hardware overhead for this purpose, however, is relatively low since most of the remaining CPU resources, such as the execution units, the branch prediction units, the instruction fetch and decode units, and the caches are either partitioned or competitively shared among the hardware contexts. Figure 2 shows a diagram of the pipeline of a SMT processor where the blue and orange colors represent instructions from different threads that are executed simultaneously on the processor. Note that some resources are replicated and privately accessed by each thread (e.g. the program counters) but others are shared, such as the caches or the execution units.

Deciding which resources are replicated and made private for each thread and which ones are shared among threads at run-time is one of the most important design issues that characterizes SMT processors. On the one hand, partitioning is the easiest solution but it does not allow a thread to use resources assigned to other threads, even if these resources are underutilized. In other words, partitioning presents a worse resource utilization. On the other hand, if resources are shared the utilization increases but also does contention.

C. Challenges of SMT-Based Real-Time Systems

In spite of the aforementioned advantages of SMT processors, which have spread their use in high performance systems, they present important issues in the context of real-time systems that should be taken into account.

One of the main drawbacks of SMT processors is that the shared resources they implement negatively affect their *predictability*, which is an important property that real-time systems should present in order to fulfill the timing constraints of critical applications. Predictability means that task results must be correct and provided within a given time domain. In order to do so, a minimum level of per-process performance must be guaranteed.

Unfortunately, ensuring predictability in SMT-based systems is difficult. Tasks running on the same physical core compete among them in the access to the shared resources and thus, interference appears. Interference causes task execution times to increase with respect to standalone execution. The impact of interference on the WCET of a particular application is variable and difficult to predict due to the high amount of core resources that are shared among co-runners in SMT architectures. Moreover, the impact also depends on the other applications that are being executed in the same core (*co-runners*) and on the sensitivity of each application to contention [10]. This means that some applications suffer a minimal performance penalty over their isolated execution while others experience serious slowdowns. For instance, the execution time of some tasks with co-runners can be doubled with respect to their execution time in isolation. Despite these issues, since SMT processors increase system productivity and power efficiency, they are being integrated in more and more systems, reaching real-time systems.

Algorithm 1 UUniFast algorithm

```
function vectU = UUniFast(n,U)
sumU = U;
for i=1:n-1
    nextSumU = sumU.*rand^(1/(n-i));
    vectU(i) = sumU - nextSumU;
    sumU = nextSumU;
end
vectU(n) = sumU;
```

Algorithm 2 UUniFast-based Workload Generator

1: **Inputs:**
2: n: number of tasks in the generated workload
3: U: workload utilization
4: UUniFast returns vector $U[1..n]$ with tasks utilizations
5: Pick n random applications from available benchmark sets as tasks
6: **for** $1 \leq i \leq n$ **do**
7: $period_i = \frac{WCET_i}{U[i]}$
8: **end for**

III. Workload Generators

Research on real-time systems requires studying the behavior of multiple workloads that present different characteristics. However, designing real-time workloads is more complex that designing workloads for evaluating HPC systems. A real-time workload is composed of tasks with their corresponding timing parameters (e.g. WCET, deadline, etc.). In this context, a realistic workload should present task deadlines long enough to allow most task execution instances to correctly complete, which does not only depend on single tasks but on all the tasks that compose the workload. Nevertheless, deadline misses should be possible to quantitatively explore the potential benefits of research proposals. Otherwise, the workload will lack of any interest for real-time systems evaluation.

Consequently, designing useful and representative workloads for real-time systems can become a tedious and time-consuming task, particularly when the number of applications grows, if the researcher needs to tune the workloads by hand to ensure certain schedulability level. To address this issue, several automatic real-time workload generators have been developed and used in the related work [11] [12] [13]. A workload generator automatically provides experimental workloads (i.e. task sets) that feed a real-time system to study its behavior.

Typically, a workload generator produces workloads composed of random tasks targeting a given theoretical processor utilization (see Equation 1). This is because, as explained above, the theoretical utilization is highly related with the number of deadline misses, which is a key performance metric for evaluating real-time systems.

A. UUniFast Workload Generator

One of the most widely known workload generators for real-time systems is UUniFast [11]. The code in Algorithm 1 illustrates the implementation of UUniFast in Matlab, as shown in the original work by Bini and Butazzo. UUniFast is used to generate task sets with a random uniform distribution of tasks theoretical utilizations.

More precisely, given a target number of tasks n and utilization U, UUniFast just provides a random set of tasks utilizations (i.e. a set of C_i/P_i ratios, where $0 \leq i < n$). Thus, to completely define the generated workload, an algorithm like Algorithm 2 must be used. This algorithm receives n and U as input parameters and gets from UUniFast the mentioned set of tasks utilizations. Then, it randomly selects n applications from the available benchmark suites in the experimental framework and, for each selected application i, calculates its period taking into account both the $WCET_i$ and the random task utilization C_i/P_i provided by UUniFast.

However, as Section V will show, in an SMT-based real-time system, the execution of workloads generated with UUniFast suffers an unexpectedly high number of deadline misses, even for relatively low processor utilizations. We found that this occurs because the UUniFast algorithm is not aware of the increase of execution time that co-runners cause due to the inter-application interference in an SMT system. In fact, the UUniFast algorithm is designed for the analysis of theoretical real-time models that do not take into account system details that increase the effective processor utilization and consequently, the number of deadline misses in some cases.

In addition, UUniFast lacks enough flexibility to specify workload restrictions that can be of interest for real-time research. For instance, workloads where some tasks have high utilizations, that is, tight periods and thus they are more likely to suffer deadline misses while the remaining tasks experience more relaxed deadlines. This happens because in UUniFast, a low target utilization value causes most task deadlines to be significantly far from the end of their corresponding WCETs, whereas a high target utilization produces deadline misses in most tasks.

B. Task Class-Based Workload Generator

The mentioned problems motivated us to design a novel workload generator aimed at SMT-based real-time systems, whose first main goal is to be more flexible than UUniFast. This is done by introducing a new concept, the *task class*. A task class c corresponds to a given rank of possible task utilizations $[MinU_c, MaxU_c]$, so tasks pertaining to the task class present similar timing constraints. The tasks in a generated workload are distributed among different task classes.

By controlling the task utilization bounds of classes and the amount of tasks pertaining to each class, a wide variety of workloads with different characteristics can be designed to simulate distinct real-time scenarios. In particular, we pursue to generate workloads with tasks presenting a wide diversity of real-time constraints, avoiding task sets where all tasks present

Algorithm 3 TCB Workload Generator

1: **Inputs:**
2: n: number of tasks in the generated workload
3: Percentages of tasks for each class
4: Utilization bounds for each class ($MinU_c$, $MaxU_c$)
5: Pick n random applications from available benchmark sets as tasks
6: Distribute the tasks among the classes complying with the input percentages
7: **for** each class c **do**
8: **for** each task i in c **do**
9: $Period_i = \frac{WCET_i}{Random\ Utilization\ [MinU_c,\ MaxU_c]}$
10: **end for**
11: **end for**
12: Return workload

either relatively relaxed or tight deadlines. In this work, for illustrative purposes, it is studied and checked the behavior of the workload generator using three different classes.

The second main goal of the proposed workload generator is to enable the designer to control the tasks that will be more affected (i.e. will be more prone to lose their deadlines) by the inter-application interference at the shared resources. This also can be done with task classes, since the utilization bounds of a task class imply a certain level of deadline adjustment.

Traditional workloads generators like UUniFast target a simple processor utilization value. However, this approach, mainly due to inter-application interference, is hard to fulfill. In the TCB workload generator we have devised a novel approach, where a range of utilization (utilization bounds) is defined to each class. Flexibility has been enhanced by specifying the percentage of tasks in the class.

The TCB workload generator requires a set of available applications or tasks. Note that the generator working principles can be extrapolated to any set of applications. The user configures the number of processes in a workload and their distribution in task classes and, as a result, the generator returns the real-time workload. That is, a list of randomly selected applications with their assigned random (within the class limits) periods.

Algorithm 3 summarizes the operation of the task class-based workload generator. Lines 2 to 4 show the input parameters that the TCB workload generator should receive to obtain a new workload. These parameters are the number of tasks of the workload, the distribution of these tasks across the configured classes, and the utilization bounds for the tasks of each class.

First, the workload generator randomly selects *n* applications from the set of available benchmarks (line 5) and distributes them among the different classes (line 6), obeying the percentage of tasks that should be assigned to each class as defined by the user. By default, we let any task to be assigned to any class, but the algorithm can be easily tuned to bind certain tasks to particular classes.

Then, the algorithm iterates trough the tasks to assign them a period (line 9). Periods are calculated as the WCET of the task divided by a random utilization, determined following a uniform distribution between the lower and upper utilization bounds defined for the task class.

Finally, the workload generator returns the list of tasks that form the devised workload with their assigned periods.

IV. Experimental Framework

We have tested the TCB workload generator in a Intel Xeon E5645 system, which is a six-core SMT processor where each core can simultaneously run up to two threads (SMT2). Each core implements private L1 (32 KB) and L2 (256 KB) caches, and all the cores share a 12 MB LLC and 12 GB DDR3 SDRAM main memory. The system has installed a Fedora Core 10 Linux distribution with kernel 3.11.4.

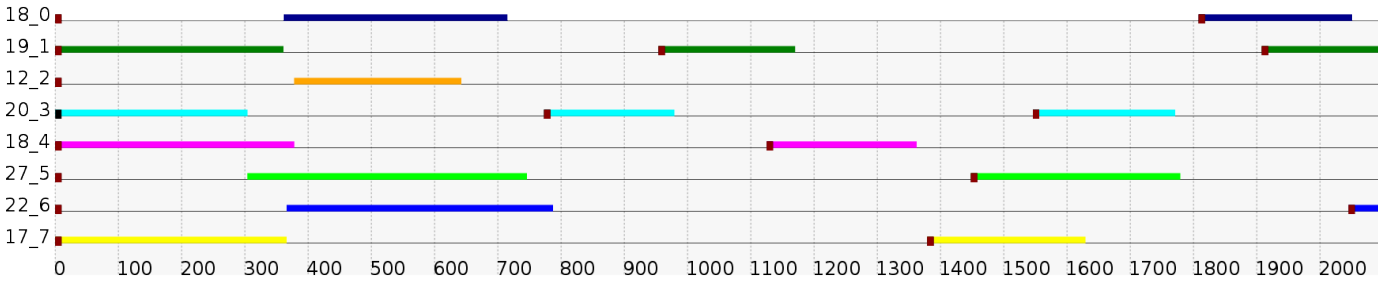
To evaluate the workloads we have extended an in-house user-level scheduler developed in previous works [10], [14] to consider the specific characteristics of real-time workloads. More precisely, we have added the deadline and period parameters to the tasks and modified the scheduler to: 1) automatically kill tasks that have not complete before their deadline when it is reached, 2) automatically activate the tasks on each period, and 3) account for real-time statistics such as deadline hits and misses for the tasks. In addition, we have implemented the Early Deadline First (EDF) scheduling algorithm [15], which is a widely used scheduling policy for real-time systems.

Finally, a set of eleven benchmarks taken from different benchmark suites has been considered as tasks. The benchmarks share two characteristics. First, they include tasks typically performed by nowadays real-time systems such as video and audio encoding and decoding, artificial intelligence processes, or image processing. Second, they are computationally intense and thus workloads composed of these tasks should run on a multicore processor with relatively high performance.

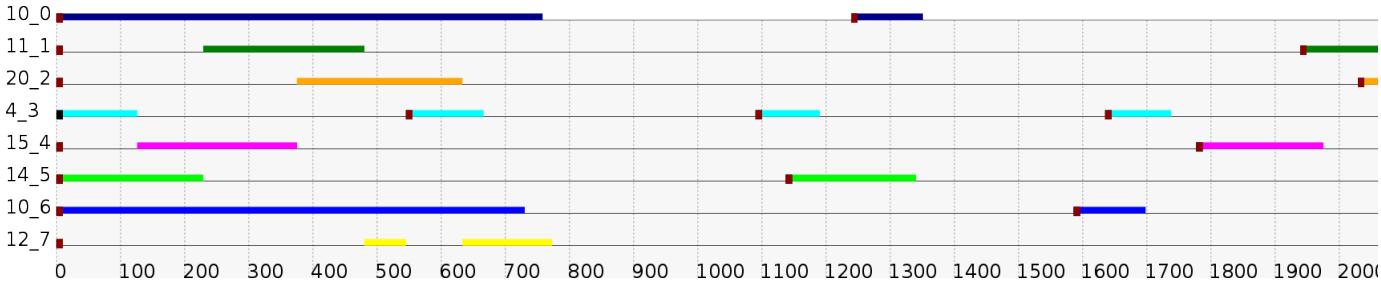
V. TCB Working Examples

This section illustrates how mixes generated by the proposed algorithm are more appropriate to evaluate current real-time systems than mixes generated with the UUniFast workload generator. For this purpose, a sample of three workloads obtained with both the UUniFast and the TCB workload generators is presented. These workloads are analyzed using two SMT2 cores of the experimental platform (i.e. the number of active hardware contexts is equal to 4) and the EDF scheduling policy.

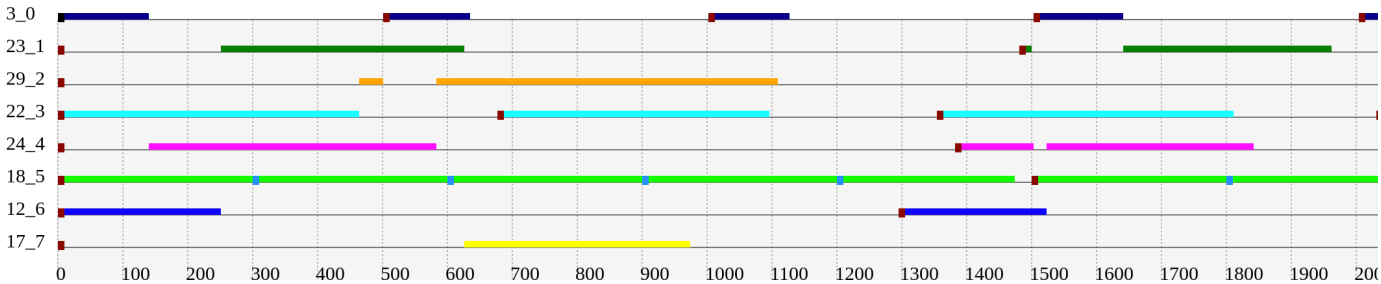
Figure 3 shows a sample execution of the three 8-task workloads obtained with UUniFast. Note that, as discussed in Section III-A, the UUniFast algorithm receives as inputs the number of tasks of the workload and the overall target utilization, but it does not allow distinct real-time constraints for different tasks. Accordingly to the number of active hardware contexts, we set the UUniFast target utilization to 4.



(a) UUniFast workload 1 (Theoretical utilization = 4)



(b) UUniFast workload 2 (Theoretical utilization = 4)



(c) UUniFast Workload 3 (Theoretical utilization = 4)

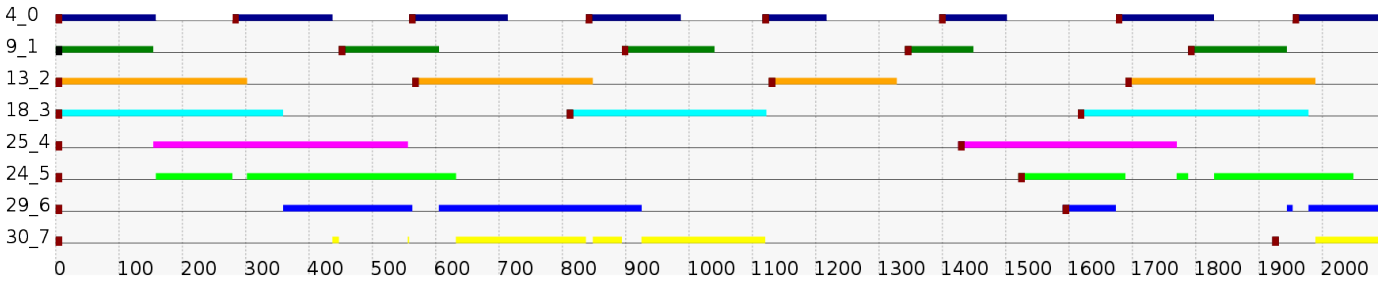
Fig. 3. Sample workloads obtained with UUniFast. X-axis represents execution time in quanta and Y-axis the applications that form the workload.

In the figure, for each task, the colored line represents the task execution and the small squares represent the deadlines. Tasks should complete their execution before a new instance of the task is launched ($D_i = P_i$). Brown squares represent a deadline hit (the task completes execution before the deadline) and blue squares represent deadline misses. Tasks are labeled as X_Y, where X refers to the application id and Y refers to the task position within the workload. The three workloads have a theoretical utilization of 4 since the UUniFast algorithm generates workloads with the exactly requested utilization.

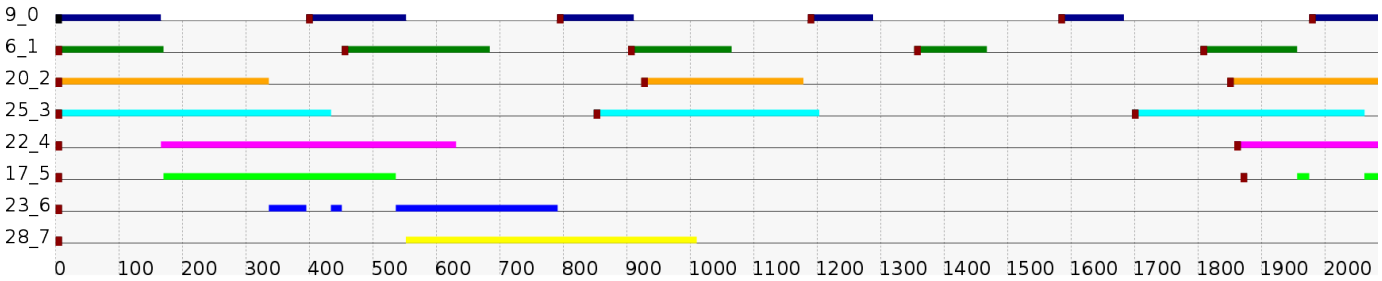
The plots illustrate the two main shortcomings that workloads generated with the UUniFast algorithm can present. On the one hand, even with a theoretical utilization set to 4, all the tasks of the first two workloads present too relaxed time constraints and tasks complete execution long before their deadline. This type of workload does not present any interest for SMT research, since the interference among applications that are running concurrently do not prevent any deadline to be fulfilled. To increase the probability of suffering deadline misses due to the inter-application interference in UUniFast workloads, the target utilization needs to be increased. However, in this case most applications suffer deadline misses, which is not representative of real life scenarios.

On the other hand, when the overall target utilization in UUniFast is higher than 1 (as it is in our scenario), deadlines can be placed too strictly, which takes to scenarios where some applications can suffer continuous deadline misses [16]. For instance, task 18_5 in the third sample workload presents too strict time constraints so it can only meet its deadline one time out of seven executions. We have checked that this problem occurs even in low interference situations. Notice that this type of workloads would not be benefited by interference-aware schedulers.

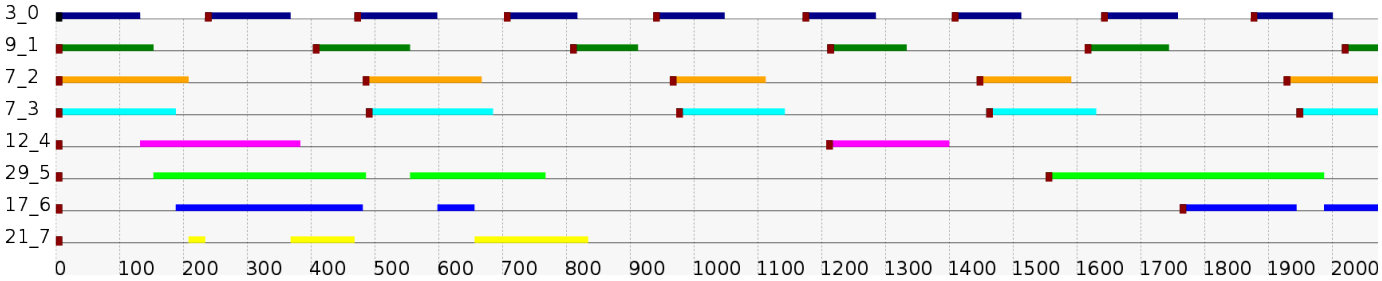
Figure 4 presents three sample workloads obtained with our TCB workload generator. The generator is configured to work with three task classes according to the following specifications: 40% of tasks belong to the first class, 50% to the second class, and the remaining 10% to the third one. Recall that task classes differ among them in the limits that restrict the random task utilization, which, at the same time, determine how tight the deadlines are. The tasks of the first class are assigned higher individual utilizations and thus will



(a) TCB workload 1 (Theoretical utilization = 3.64)



(b) TCB workload 2 (Theoretical utilization = 3.46)



(c) TCB workload 3 (Theoretical utilization = 3.49)

Fig. 4. Example of workloads obtained with TCB. X-axis represents execution time in quanta and Y-axis the applications that form the workload.

present tighter periods. To this end, the upper and lower bounds of the first class were set to 0.95 and 0.65, respectively. On the contrary, the tasks of the third class present the lowest utilization and hence more relaxed periods, setting its lower and upper bounds to 0.20 and 0.33, respectively. Tasks in the second class receive an utilization falling in between the first and third classes. Notice that by distributing tasks into these categories we generate workloads that will have tight, medium, and relaxed deadlines.

It can be observed that the tasks in these plots present heterogeneous time restrictions depending on their class. For instance, in the first workload, task 4_0 and 9_1 present shorter periods and thus they are more exposed to deadline misses due to interference. In contrast, other tasks like 30_7 present wider periods and should not be a problem for the scheduler.

Unlike the workloads obtained by the UUniFast algorithm, the workloads generated with our proposed TCB generator do not match an exact utilization. The three generated workloads present theoretical utilizations of 3.64, 3.46, and 3.49. Despite their lower theoretical utilization, these workloads are more meaningful for SMT-based real-time systems research, since TCB includes applications potentially affected by key interference-related issues that can be solved by a smart scheduler. For example, one possible strategy could be to place applications that cause low interference in the same core as applications that are sensitive to interference. In this way, the impact on the performance of the interference-sensitive application is lower, which reduces the amount of deadline misses.

VI. Conclusions and Future Work

The complexity of tasks requiring real-time execution has been increasing lately and it is expected to keep increasing in the next years. Multimedia systems and self-driving cars are only two examples of scenarios where both real-time and high performance are required. To cope with the performance requirements, real-time systems are nowadays moving to multicore processors and will likely move to SMT processors in the next years. Research on real-time systems with these architectures is therefore expected to grow.

In this paper we have proposed the task class-based (TCB) workload generator, which aims at easing and automating the process of designing meaningful real-time workloads for

these systems. Unlike previous workload generators such as UUniFast, the TCB generator is designed aimed at helping researchers investigate on sharing resources and meeting deadlines in SMT multicore systems. A novel feature of TCB is the concept of task classes that allows setting the percentage of tasks with deadline tightness bounded within defined thresholds. TCB has been checked and compared to UUniFast. For this purpose multiple workloads have been generated and their characteristics analyzed. It has been proven that TCB provides more meaningful real-time workloads for SMT-based real-time systems research.

As a future work, we plan to design different scheduling policies to minimize the performance variability and maximize the throughput of real-time systems with SMT multicore processors. The TCB workload generator will be used to easily obtain meaningful workloads to evaluate the different scheduling algorithms on the scenarios under study.

ACKNOWLEDGMENTS

This work has been has been funded by the Spanish Ministerio de Economía y Competitividad (MINECO) and Plan E funds, under grants TIN2015-66972-C5-1-R and TIN2017-92139-EXP, and ExaNest project funded by the European Union's Horizon 2020 under grant agreement No 671553. Josué Feliu has been supported through a postdoctoral fellowship by the Generalitat Valenciana (APOSTD/2017/052).

REFERENCES

[1] L. Dozio and P. Mantegazza, "Linux real time application interface (rtai) in low cost high performance motion control," 2003.

[2] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun, "Towards fully autonomous driving: Systems and algorithms," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, 2011, pp. 163–168.

[3] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 149–163.

[4] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *10th IEEE International Conference on Computer and Information Technology*, 2010, pp. 1864–1871.

[5] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore systems," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 57–68, 2009.

[6] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiones, M. Gerdes, M. Paolieri, J. Wolf, H. Cass, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische, "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

[7] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 392–403, May 1995.

[8] J. Burns and J.-L. Gaudiot, "Smt layout overhead and scalability," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 2, 2002, pp. 142–155.

[9] D. B. Y. Li, K. Skadron and Z. Hu, "Performance, energy, and thermal considerations for smt and cmp architectures," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 71–82.

[10] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Bandwidth-aware online scheduling in smt multicores," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 422–434, Feb 2016.

[11] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[12] M. Cirinei and T. P. Baker, "Edzl scheduling analysis," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, 2007, pp. 9–18.

[13] R. I. Davis and A. Burns, "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," *Real-Time Systems*, vol. 47, no. 1, pp. 1–40, Jan 2011.

[14] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic Job Scheduling on the IBM POWER8," in *Proceedings of the 22nd International Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 669–680.

[15] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," in *Journal of the ACM*, vol. 20, no. 1, 1973, pp. 46–61.

[16] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS workshop at the Euromicro Conference on Real-Time Systems*, 2010, pp. 6–11.